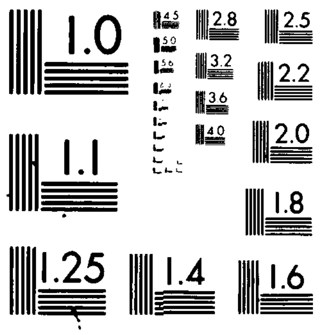
AD-A098 755 GENERAL RESEARCH CORP SANTA BARBARA CA F/G 9/2
COBOL AUTOMATED VERIFICATION SYSTEM: STUDY PHASE.(U)
MAR 81 R MELTON, G GREENBURG, M SHARP F30602-80-C-0101
UNCLASSIFIED RADC-TR-81-11 NL

1 OF 1
AD
A098755

END
DATE
FILMED
8-81
DTIC


1.0
4.5
5.0
5.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

LEVEL II (12)

AD A098755

RADC-TR-81-11
Final Technical Report
March 1981

COBOL AUTOMATED VERIFICATION SYSTEM: STUDY PHASE

Richard Melton
Gary Greenburg
Michael Sharp

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

81 5 11 023

RADC-TR-81-11 has been reviewed and is approved for publication.

APPROVED: [illegible]

LAWRENCE M. LOMBARDO
Project Engineer

APPROVED: [illegible]

JOHN J. MARCINIAK, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

17

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 18 RADC-TR-81-11	2. GOVT ACCESSION NO. AD-A098755	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) COBOL AUTOMATED VERIFICATION SYSTEM: STUDY PHASE		9 5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Mar[redacted] — Sep[redacted] 80
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Richard Melton Gary Greenburg Michael Sharp		15 8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0101 new
9. PERFORMING ORGANIZATION NAME AND ADDRESS General Research Corporation PO Box 6770 Santa Barbara CA 93111		16 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 25310205 [illegible]
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIE) Griffiss AFB NY 13441		11 12. REPORT DATE Mar[redacted]981
		13. NUMBER OF PAGES 84 12 [illegible]
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Lawrence M. Lombardo (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Software Testing
Computer Software Verification
COBOL 68
COBOL 74
Software Development Tool

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents the results of a study to specify the required capabilities and high-level design of an automated tool to support the testing and verification of COBOL software systems. Included is a state-of-the-art review of software testing and verification with emphasis on techniques applicable to COBOL programs.

[illegible] [illegible]

ABSTRACT

The COBOL language, and automated software testing tools, have been studied in order to design an Automated Verification System for COBOL. The proposed functions and design of the system are summarized in this report. Details of the system are presented in the "CAVS Functional Description" and "CAVS System/Subsystem Specification." To provide a perspective for the capabilities of the proposed system, this report contains a critique of the COBOL language, a description of methods of software testing, and a characterization of errors in COBOL. Considerations for future capabilities of the system are also outlined.

CONTENTS

FIGURES

TABLES

The purpose of this contractual effort was to determine and specify the required capabilities for an automated testing and verification system for COBOL software systems. The effort provided a significant review of the state-of-the-art of software testing and verification, with emphasis placed on techniques applicable to COBOL programs. The resulting capabilities were specified in two separate documents - a Functional Description and a System/Subsystem Specification, which will be utilized during the implementation phase of the effort. The availability of an automated testing and verification system for COBOL is significant in that it will enhance Air Force software development capability and result in a more cost-effective and reliable product. This effort was responsive to the objective of the RADC Technology Plan, TPO 4G3, "Software Development and Test Tools."

[illegible]
LAWRENCE M. LOMBARDO
Project Engineer

1 INTRODUCTION

General Research Corporation is under a two-phase contract with Rome Air Development Center to design and implement an automated tool to assist in the development, testing, verification, and maintenance of COBOL software.

Phase 1 of this contract was (1) the study of the COBOL language and recent advances in the testing and verification of computer software, with emphasis on techniques applicable to COBOL programs, and (2) the development of a functional description and system/subsystem specifications for the tool.

Phase 2 will be the implementation, testing, and user training period.

This final report on Phase 1 describes our progress through the study phase, presents highlights of the functional description and system/subsystem specifications (which are given in detail elsewhere), reports our findings and conclusions, and proposes future capabilities for consideration.

General Research Corporation's Software Quality Department made an in-depth study of COBOL: the structure of the language, its history, the type of installations which use it, the reasons they chose it, the problems they have encountered in programming in the language, COBOL's limitations and advantages, and the computers on which it is being used. We examined a wide cross-section of COBOL tools available on the market today.

We found that (1) the majority of available commercial COBOL software tools were designed for IBM systems, since IBM commands such a large portion of the market, and (2) although many of the tools available today offer good testing features, no one offers more than one or two of the available methods. To put together a comprehensive software quality package would require purchasing several tools, each with its own command language and eccentricities. Further, communication between the tools would be quite difficult; not all the tools are available for a particular brand of computer; and many of the tools have poor documentation. Also, since they are generally written in assembly language or a scientific language such as Fortran or PL1, maintenance would be difficult.

The COBOL Automated Verification System (CAVS) is intended to be a comprehensive software-quality package that does not suffer from these deficiencies. CAVS will be written in American National Standard COBOL - 1974. CAVS will accept for testing analysis any COBOL program written in ANSI-COBOL 1968 or ANSI-COBOL 1974 for the Univac, Honeywell, or DEC VAX computer systems. The COBOL compilers for these computers are not exactly alike; therefore CAVS will be written in a subset of ANSI-COBOL 1974 which is compilable on all three. By this method, CAVS will be made portable rather than having to be written in three versions.

CAVS will be implemented with the well-proven methodology of structured, modular design. This will permit easy maintenance and modification in the event of later enhancements, changes in the computer system, or changes in the design of CAVS.

In short, CAVS will be a comprehensive collection of the most current techniques for software testing and program development, organized into one tool and requiring one easy-to-execute command language.

DOCUMENTS DEVELOPED DURING PHASE 1

1. Functional Description
2. System/Subsystem Specification
3. Final Report - Study Phase
4. Draft Preliminary User's Manual

DOCUMENTS TO BE DEVELOPED DURING PHASE 2

1. User's Manual
2. Maintenance Manual
3. Test Plan
4. Program Specification
5. Final Report
6. Training Material
7. Testing Report

2 TIMETABLE

The following chart presents the schedule of activities for Phase 2 and the DMA option of the COBOL Automated Verification System project.

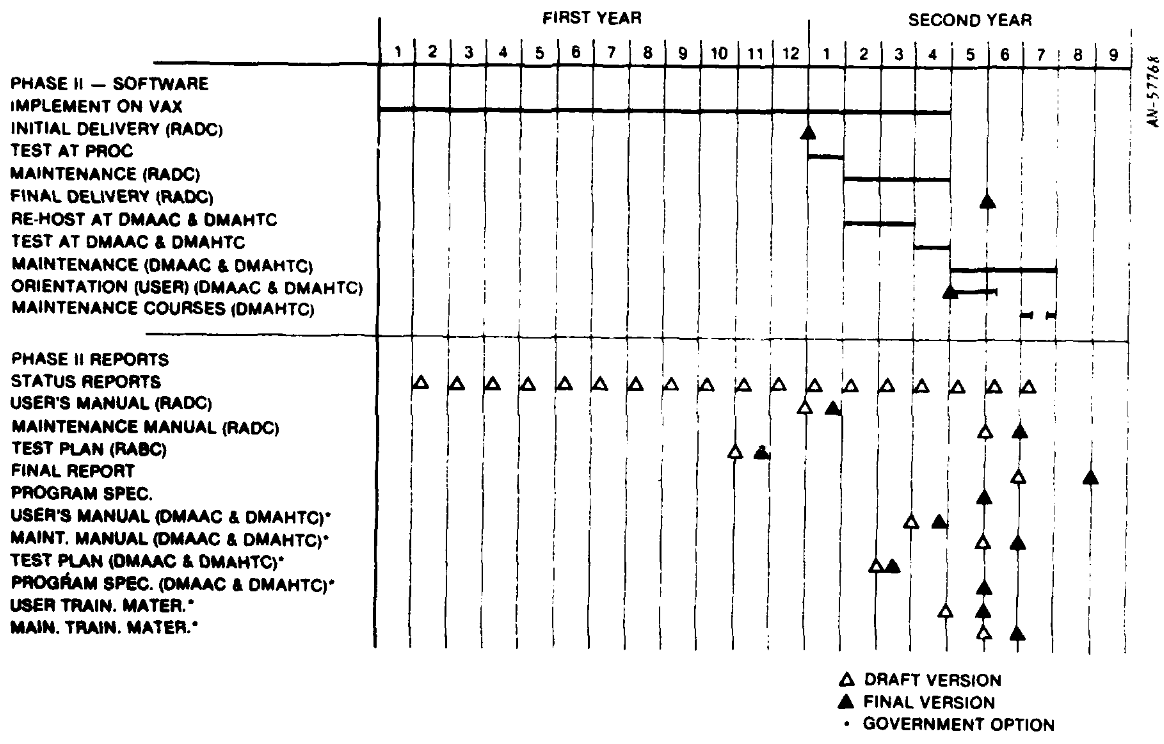


Figure 2.1. Proposed Implementation Schedule for CAVS

3 THE COBOL LANGUAGE

3.1 HISTORICAL BACKGROUND

During the 1950s as manufacturers entered the computer business, each one developed its own computer programming language for its own machines. Program portability was non-existent and it became increasingly difficult for programmers to be mobile. The Federal Government, the largest user of computers, became concerned about the need for a "common" programming language for business applications of data processing.

In 1959 the original specifications for the COBOL language were drawn up by a group of computer users and manufacturers. The first documentation was distributed in April 1960. The early 1960s brought several revisions to COBOL, each one making the language less "common" to the different computers in use at that time.

Again, the manufacturers assembled and developed a new, "standard" COBOL called American National Standard COBOL (ANS COBOL). The new language gained widespread acceptance in the US business sector and throughout the world. The further development and definition of COBOL is the function of the CODASYL (Conference On Data System Language) COBOL Programming Language Committee.

The standard of the language in the US (an extensive subset of the full CODASYL COBOL definition) is the American National Standard COBOL, X3.23-1974, as approved by the American National Standards Institute (ANSI). This has replaced the previous ANS COBOL X3.23-1968. A new ANS COBOL should be completed before the end of 1981.

3.2 ANS COBOL - 1968 AND 1974

ANS COBOL-1968 was the first effort by the CODASYL-ANSI group to define COBOL as a programming language to be a standard throughout the world on all those computer systems which chose to conform to its guidelines.

After a few years of working with these guidelines, the group found that the main skeleton of the language was homogeneous among most of the participating computer users. However, due to lack of specificity in some guidelines and the differences in the hardware and design of the computer systems, some sections of the language were markedly different from one machine to another. Gathering all this information, the CODASYL group assembled again and produced a new set of more specific guidelines and released ANS COBOL-1974.

The ANS-74 version of COBOL was created to (1) delete sections that hindered efficient coding or standardization, (2) add sections to enhance COBOL's capabilities, and (3) resolve differences or ambiguities created by the different compiler manufacturers' versions of the language. The third item was accomplished by being more specific in the wording of those sections of the Standard, and by requiring the compiler manufacturers to adhere more closely to the specifications.

A great improvement over ANS COBOL-1968 had been realized. As the computer manufacturers finished their versions of COBOL according to ANS COBOL-1974 specifications, and the different versions were compared, it was found that there were far fewer differences this time, although they were not identical because of the inherent differences mentioned above. Each manufacturer respected the guidelines and any serious deviations were noted as extensions to the ANS.

Our study has had three objectives: (1) determine the differences between 1968 and 1974 ANS COBOL to make certain our AVS will recognize both versions, (2) determine differences between the computer manufacturers' dialects for the same reason, and (3) develop a skeleton COBOL (Appendix B) common to Univac, Honeywell, and DEC VAX machines in which to write our AVS.

<u>COBOL Dialect Differences</u>. A truly portable COBOL AVS must recognize not only the two standards, 1968 and 1974, but also the dialects of different computers. Design and operational differences exist between computers, and although each satisfactorily compiles a program which meets the requirements set by the Standards committee, it will not compile a program which uses another manufacturer's enhancements to the standard. Users at each installation make use of these enhancements.

3.3 THE NATURE OF COBOL

COBOL was created to solve the special data processing problems of the business world. The language was not designed to solve complex mathematical or scientific problems or to facilitate number-crunching computer analysis, but rather to expedite the handling of everyday business affairs with great speed and accuracy. COBOL was created to process accounting, payroll, inventory, tax, and data base maintenance programs in a manner which allowed efficient use of large data files of information. Most business programmers are not highly-trained scientists, so the syntax or wording of the language was designed to be as similar to everyday English as possible. Importance was placed not so much on features such as mathematical functions and speed of calculations as on efficient input and output of large data files stored on magnetic tape or disk.

A typical COBOL system could be described as a program (with usually not more than five subroutines) of approximately 1000-5000 lines of code. The program is designed to create or update a large file of data and then produce update and error reports. Mathematical operations are mostly arithmetical: add, subtract, multiply, and divide. Data is read into the computer usually from secondary storage such as magnetic tape or disk, updated or selected by conditional statements, then reformatted for printing or storage.

A business-oriented data processing problem can be broken down into four distinct groups of logically related information:

1. Identification of the type of problem (accounting, payroll, etc.)

2. The data processing environment in which the problem is to be solved (the computer and peripheral equipment needed to solve the problem).

3. Description of the data to be processed, the format of the data in a record, and the format of records in a file. In addition, the organization of the files must be described, and the processing mode used must be stated.

4. The procedure(s) by which the data is to be processed to solve the problem.

The COBOL language is structured to accomodate these four groups of logically related information, in four named divisions: IDENTIFICATION, ENVIRONMENT, DATA, and PROCEDURE. Every COBOL program must contain these four sections, and the structure of the language dictates a specific format to be used to describe the information in each.

COBOL Errors. Martin[1] analyzed the type of errors occurring in delivered COBOL programs in an attempt to evaluate the effect of complexity on error frequency. Commercial and research COBOL programs running on Honeywell 6060 and NCR 8200 systems were tested by a source program analyzer. In summary, he found the following types and frequencies of errors.

Error Type	Percentage of Total Errors
Logical	25
Data Handling	18
Input/Output	16
Computational	11
Interface (Subprograms)	9
Data Base	7
Other	14

Analyzing this table, we see that 36% of the total errors were those types of errors which occur uniformly among most computer languages, the logic and computational errors. However, the errors which arise from the data handling, input/output, and data base sections, typically COBOL errors, account for 41% of the total errors.

[1] C. E. Martin, A Model for Estimating the Number of Residual Errors in COBOL Programs, Ph.D. Thesis, Auburn University CI-77-97.

An internal Honeywell study[1] on properties of COBOL programs compared the static and dynamic characteristics of commercial programs. Researchers used a source program analyzer to count the occurrences of critical COBOL keywords. Dynamic performance was monitored by executing instrumented versions of the same programs. The study concluded:

- The most frequently coded source statement type was MOVE, followed by GO TO, IF, WRITE, ADD, and PERFORM, in that order.

- The most frequently executed statement was IF, followed by GO TO, ADD, MOVE, PERFORM, and READ in that order. IF statements made up almost half of the total statements executed.

- The static and dynamic nature of programs differs greatly. Simple inspection and static analysis will not yield a valid picture of run-time behavior.

The large number of IF-statement executions appears consistent with the observation that most commercial COBOL programs input large amounts of data to be either tested by conditional statements for certain properties, selected for modification, or output in reports, rather than perform extensive arithmetic operations.

[1] R.J. Chevance and T. Heidet, "Static Profile and Dynamic Behavior of COBOL Programs", SIGPLAN Notices, Vol. 13, No. 1, April 1978.

4 SOFTWARE TESTING AND VERIFICATION

4.1 INTRODUCTION

One of the goals of this phase of the CAVS project was, through research and analysis of current tools and techniques, to develop a methodology for systematically and comprehensively testing COBOL software.

Usually, COBOL software is tested only according to its developer's intuitions, if it is tested at all. Since the reliability of software is at least partially dependent upon the thoroughness of its testing, increased testing therefore contributes to increased reliability.

Simple computer programs can be comprehensively tested without difficulty. When computer software becomes complex, usually by length of program or number of paths possible so that human intuition is inadequate to deal with its subtleties, the testing activity must be based on a systematic and rigorous methodology. Most COBOL software systems are lengthy or complex, so that the advantages of an automated verification system become pronounced and desirable.

Approaches to Software Quality. The computer science community has recognized the problems concerning software correctness and has been developing systematic approaches to increase the reliability of software and simultaneously reduce the overall cost of producing it.

"Synthesis" techniques generally try to increase software quality by keeping software problems from happening in the first place. For example:

- Structured programming disciplines reduce the complexity of software (and thereby enhance its quality and reliability) by constraining the control structures of the programming language used.

- Chief programmer teams assign a talented person entire responsibility for all aspects of a software system, including its ultimate effectiveness and reliability.
- Software design methodologies such as "top down" or "bottom up" systematize the production of software and thereby improve the quality of the programs.

The alternative, to deal with software which has already been developed (or is in the final stages of development), involves two primary "analysis" approaches:

- Program proof demonstrates the correctness of programs by treating them as if they were mathematical theorems. An automated theorem prover is often used to assist in the construction of proofs.
- Automated Verification Systems (AVS) increase the practical reliability of software by increasing the level of "testedness" achieved.

Limitations. Although advances are being made, program proving through logical or mathematical theorems is impractical today for programs of any size. Further, there is still discussion as to whether this mode of testing is "more correct" than other methods.

An Automated Verification System, however, is a valuable tool. The role of the AVS is to assure that software testing meets some criterion of completeness. Comprehensive exercise of a software system does not guarantee that it is error-free, but practical experience indicates that thorough exercise will locate a very high proportion of errors. Hence, testing with an AVS as an approximation to full program verification, along with proper system design, is a practical and valuable methodology.

4.2 TESTING METHODS

The common concept uniting this study is that software verification is a combination of separate techniques that, when applied together, form a good base methodology for testing. These techniques are (1) systematic design methodologies, (2) documentation, (3) static testing, and (4) dynamic testing and performance measurements.

4.2.1 Systematic Design Methodologies

It is imperative that software design be efficient, logical, and correct. Bad design increases programming time, programming errors, execution time, and maintenance frustrations. The technique of structured, modular design has been shown by working experience to be of great value in reducing these problems.

If a system has been designed and implemented in a structured fashion (top-down, bottom-up) using structured constructs and dividing program tasks into modules, the design, coding, and testing can be done in small steps. Further, enhancements or changes to the system can be done with ease and efficiency.

Most COBOL programs will need to be changed as the needs of the user change. It is therefore valuable to design a program so that it can be modified or maintained. Good planning and structure early in the design phase plays a large role in this. An automated verification system should encourage the use of structured programming and increase the value of the program written in modular form.

4.2.2 Documentation

Because maintenance of COBOL software has become such a large concern, there is a need for good comprehensive documentation. Personnel turnover, constant modification of programs, and high cost of programming time make it imperative that documentation of the system, from design through maintenance, be up-to-date and complete.

Examples of automated documentation include cross-reference listings, program management systems, and printed reports such as those produced by DAS and DCD II,[1] or by an automated verification system such as FAVS and JAVS.

4.2.3 Static Program Analysis

One class of methods for software quality enhancement can be categorized as "static analysis". These methods scan the source text of a program for errors in syntax and semantics which can be detected without running the program on a computer. They provide consistency checking and documentation about the definition, reference, and communication of data within the program. They identify programming constructs which may be legal but risky; and they provide global, organized information about the identifiers used in the program. Static analysis expands upon the sort of diagnosis performed by a typical compiler. In general, static analyzers are most useful in debugging.

4.2.4 Dynamic Program Analysis

Two basic types of dynamic program analysis are: analysis of statement-level behavior and analysis of execution coverage. Both are well-known, general-purpose testing aids.

Statement-level Analysis. In statement-level dynamic analysis, all program statements are instrumented in order to obtain detailed information concerning the program's internal behavior. This technique produces information that is more detailed and more closely related to the source program information than such earlier techniques as hardware monitoring, software monitoring ("snapshots"), and simulation techni-

[1] DAS and DCD II are products of CGA Computer Associates of Rockville, Maryland.

ques. Typically, a statement-level preprocessor automatically augments each source program statement with a "software probe" -- added statements or the invocation of a subroutine which takes measurements while the program is running. These measurements usually include the values of selected program variables and the number and types of branches taken.

When the program terminates, summary reports are printed which show the ranges of the program's intermediate variable values, which branches were taken and with what frequency, and which statements in the program were not executed.

Execution Coverage Analysis. This technique gathers information on the run-time sequencing of a program and the flow of control among the programs that make up a programming system. This sequencing information can be represented at various levels of detail. At the lowest level it may be a trace of the statements executed by a program when run with a particular testcase, or the sequence of branches executed by the program. At a higher level, the actual program flows traversed by the program may be collected or, at a still higher level, the dynamic calling sequence of procedures and subroutines in a programming system may be monitored.

The technique for implementing execution coverage analysis is the same as that for statement-level analysis; that is, placing software probes in the programs at the level at which monitoring information is to be gathered. The added statements are simply invocations of run-time auditing procedures which record which procedure and which control sequence or statement is being executed at the time of monitoring. A post-processor can then reproduce the dynamic flow of control through a single program or a group of programs at whatever level is desired. This information is useful in determining which control flows and procedures were exercised by which test cases as a guide to what testing remains to be done.

4.2.5 Existing Methods And Procedures

We have looked at the types of methodology desirable for a comprehensive verification system in the preceding section. Below are the methods and procedures, along with the companies creating them, which are now in use for software verification

For the most part, software verification is still a manual process. Tools and techniques exist, but this area of software engineering is in its infancy. Most of the tools and methodologies have severe restrictions or require highly-skilled persons to make their application successful.

Requirements. Requirements state what a computer system should do from the user's viewpoint. Manual systems exist which aid system decomposition via graphical techniques (SADT from SofTech and AXES from Higher Order Software) and which label requirements so the labels can be inserted in the design and code (THREADS from Computer Sciences Corporation) for tracing requirements to the code.

Specification. At least two languages and tools exist for stating detailed specifications (Requirements Specification Language - RSL - from TRW and SPECIAL from SRI). Both provide a rigorous means of stating specifications which can be used to detect inconsistencies. Both are expensive to use and are best utilized on small programs only.

HIPO (Hierarchy plus Input-Process-Output) charts are a manual means of stating software specifications in the context of program structure.

Design. There are many design methodologies based upon decomposition, structure, data relationships, and top-down and bottom-up development. There are also systems and languages such as Process Design System (PDS from System Development Corporation) and Process Design Language (PDL). PDL is a control-structure keyword recognizer.

Functional and Performance Testing. Manual, functional, and performance testing are assisted by deriving data from HIPO charts, using simulations, obtaining execution-time intermediate-value printout, and running stress or boundary tests. Boundary, or special value testing, is a strategy which exercises a program using certain values important to the control flow of the program. Predicates of logical expressions, and values which activate one or more conditions in a complex logical expression, are good candidates for special value tesing. Stress testing requires that the tester manually identify areas in the program which are critical to its function. These areas are then subjected to intensive testing using special values and other methods.

Functional testing treats functional components of a program as separate programs, with their own input, output, and processing requirements. Assertions can be used to determine if the requirements for each of these functions are being met. Using assertions, the execution-time behavior of these functions can be checked for:

- Inconsistencies between the specified and actual contents of variables
- Time required to execute a function
- Contents of passed parameters upon entering and leaving a function
- Changes in a function's behavior when the input values are systematically changed (as in the case of General Research's Adaptive Tester).

Structure-based Testing. This testing concept has been very popular for providing a measure of testing completeness, test data generation, error location, and finding structural anomalies. There are a number of automated tools which perform branch testing (RXVP, JAVS, FAVS, SQLAB, and TAP from GRC, NODAL from TRW, PET from McDonnell Douglas, Test Coverage Analyzer from Boeing) or execute user-specified sequences of statements (SADAT from Kernforschungszentrum Karlsruhe GmbH).

Algorithms are being developed to circumvent the impossible goal of testing all control paths in a program. Some of these techniques are (1) identifying strongly-connected components of a directed graph (Tarjan, Ramamoorthy), (2) partitioning the program graph into sub-schemes which are single-entry/single-exit structures (Sullivan), (3) identifying strongly-connected subgraphs which are single-entry/multiple-exit, called intervals (Hecht and Ullman), and (4) partitioning the program graph in terms of its iteration level, called level-i paths (Miller).

Manual structure-based testing can be assisted by deriving decision tables (Goodenough and Gerhart) and choosing input data accordingly.

Structural anomalies such as dead code, potential infinite loops, and infeasible paths can be determined by some current AVS tools (ATDG from TRW, SADAT, JAVS).

Consistency Checking. The most common techniques used to determine the consistency of variables and interfaces are:

- Adding assertions that define expected use (SQLAB from GRC, ACES from UC Berkeley)
- Employing static analysis (AMPIC from Logicon, DAVE from University of Colorado, FACES from UC Berkeley, RXVP, FAVS, and SQLAB from GRC)

- Using data flow analysis to find uninitialized variables and interface inconsistencies (DAVE, RXVP, SQLAB)

Test Data Generation. A great deal of research energy has been expended on developing test data generators. So far, these systems (such as ATTEST at the University of Massachusetts) are still research tools and have had to back off from original goals. Other tools such as test harnesses or the Adaptive Tester require input boundaries and invariances between variables to be specified.

For manual test data generation, Howden suggests that input data be chosen to reflect special values for the program. Ostrand and Weyuker suggest deriving data in two phases based upon likely errors for the particular program's function and likely errors for the control structures used in the program.

Formal Verification. Automated formal verification systems (EFFIGY from IBM, PROGRAM VERIFIER from USC/ISI, SID from the University of Texas at Austin, SQLAB from GRC, SELECT from SRI) take user-supplied assertions (called verification conditions) usually at each branch, and symbolically execute them. The systems attempt to prove each verification condition as it is symbolically executed. The process involves simplification of inequalities and, in the case of interactive provers, the input of occasional rules to aid simplification. Formal verification is still reserved for small programs. Most of the implemented systems are based on LISP.

Program Modification. Tools which utilize a database system and save interface descriptions or other such system-wide information can be helpful to support program modification and maintenance activities. Valuable information for these activities are module interaction reports, detection of global changes, and local updates. Some of the tools that provide this assistance are the Boeing Support Software, SID, JAVS, FAVS, and SQLAB.

Documentation. Automatically-generated reports which provide information about program structure, calling hierarchy, local and global symbol usage, and input and output statement location are very useful during program development, testing, and maintenance. Most AVS tools provide some or all of these reporting capabilities.

4.3 SOFTWARE TOOLS

GRC made an extensive study of software tools available today both for COBOL and for other languages in order to obtain a balanced view of the techniques and options that are offered.

COBOL is the most popular computer language in use today. There are many tools available to the COBOL programmer and analyst. The tools we investigated which were not applicable to COBOL were nevertheless useful in providing new techniques to consider.

- There are a number of text-editors such as MENTEXT, University of Maryland's editor for the Univac 1100 series, and DEC SOS. Some of the editors are powerful, others have full-screen editing capabilities and program reformatting features.
- Object-code optimizers such as CAPEX's OPTIMIZER are useful tools which reduce core requirements, eliminate unused code, reformat the compiler listing, and permit faster execution time on IBM S/370 systems.
- Tools which assist in testing and debugging (CAPEX Analyzer/Detector, FAVS, RXVP80, QUALIFIER) are available for certain computer systems.
- Additional tools are test data generators (PRO/TEST,DATA-MACS), instrumentation packages (QUALIFIER, FAVS, JAVS, PET), and data cross-reference and documentation packages (DAS, DCDII).

Many of these tools perform worthwhile functions and serve a selected market well. However, as we researched the tools, the following disadvantages presented themselves:

- Most are oriented to IBM and IBM-compatible hardware, and some are operating-system dependent.
- Some require modification of the software for system 'fit'.
- Few of the tools actually support and encourage structured programming.
- There are many vendors, each offering a tool for a specific function. Tool command languages differ; obtaining a comprehensive tool requires procuring many packages, and learning many operating languages and methods of utilization.
- Program debugging still requires extensive use of core dumps.
- Most of the software tools which incorporate static analysis and instrumentation testing have been developed for Fortran or another scientific language, usually at research facilities (PACE at TRW, PET at McDonnell Douglas, FAVS and JAVS at General Research). There are very few COBOL static analysis or instrumentation tools.

Table 4.1 lists the tools we have examined, and in some cases used, in the study.

Many of the non-COBOL tools seem to have originated as research projects and, as a result, perform general program analyses which often include building a database and a program graph. This broad base of information allows these tools (with some overhead expense) to be extended in capability.

Table 4.1 SOFTWARE TOOLS EXAMINED IN STUDY

TOOL SYSTEM	DEVELOPER	LANGUAGE	COMPUTERS	CAPABILITIES OF TOOL
ACES	UNIV. OF CA. BERKELEY	FORTRAN	CDC 6600 IBM 360 UNIVAC 1108	CROSS-REFERENCE ASSERTION RANGE CHECK CODING STANDARDS VIOLATIONS DO LOOP VIOLATION
COBOL REFORMATTER	COMPUTER ASSISTANCE, INC. STANFORD, CT	COBOL	IBM, BURROUGHS CDC, HONEWELL, NCR	REFORMATS DATA AND PROCEDURE DIVISIONS
DAS DATA ADMINISTRATION SYSTEM	CGA COMPUTER ASSOCIATES ROCKVILLE, MD	COBOL	IBM 360 UNIVAC 1100	AUTOMATED DATA CROSS-REFERENCING REPORTING COPY TEXT AND LINKAGE SECTION CROSS-REFERENCE REPORTING
DATAMACS	MANAGEMENT AND COMPUTER SERVICES, INC. MALVERN, PA	COBOL	IBM 360/370	TEST DATA GENERATOR
DCD II DATA CORRELATION AND DOCUMENTATION SYSTEM	CGA COMPUTER ASSOCIATES ROCKVILLE, MD	COBOL	IBM, DOS, HONEWELL, GCOS UNIVAC, EXEC8	LAYOUTS OF FILES, RECORDS AND WORKING STORAGE CROSS REFERENCE LISTING OF FILES RECORDS AND DATA ITEMS ENTRY/EXIT POINT HISTORY WITHIN PROCEDURE DIVISION
EASYTRIEVE	PANSOPHIC SYSTEMS OAK BROOK, IL	OS	IBM 360/370 UNIVAC SERIES 70	INFORMATION RETRIEVAL AND REPORT GENERATION SYSTEM TO GENERATE KEYED REPORTS FROM INPUT FILES
FACES	UNIV. OF CA. BERKELEY	FORTRAN	CDC 6400 IBM 360 UNIVAC 1108	CROSS-REFERENCE SET/USE VIOLATION DO LOOP VIOLATION COMMON BLOCK VIOLATION

Table 4.1 (CONTINUED)

TOOL SYSTEM	DEVELOPER	LANGUAGE	COMPUTERS	CAPABILITIES OF TOOL
FAVS	GENERAL RESEARCH CORPORATION SANTA BARBARA, CA	FORTRAN	CDC6400 VAX 11/780 UNIVAC 1100	STATIC ANALYSIS BRANCH INSTRUMENTATION AUTOMATED PROGRAM ANALYSIS DOCUMENTATION BRANCH EXECUTION COVERAGE REACHING SET GENERATION STRUCTURING FORTRAN INTO DMATRAN
FORMAT	K AND A SOFTWARE PRODUCTS DALLAS, TX	COBOL	IBM 360/370	AUTOMATED FORMATTING WITH DATA AND PROCEDURE DIVISION INDENTATION EDIT CAPABILITY
JAVS	GENERAL RESEARCH CORPORATION SANTA BARBARA, CA	JOVIAL J3	CDC 6400 HIS 6180	BRANCH AND STATEMENT EXECUTION COVERAGE BRANCH AND MODULE TRACING AUTOMATED PROGRAM ANALYSIS DOCUMENTATION ASSERTION VIOLATIONS REACHING SET GENERATION
J73AVS	GENERAL RESEARCH CORPORATION SANTA BARBARA, CA	JOVIAL J73	ITEL AS/6 (IBM SYSTEM/370) DEC SYSTEM 20 CDC CYBER 175	ONLINE AND BATCH OPERATION STATIC AND DATA FLOW ANALYSIS TEST HISTORY REPORTING PATH IDENTIFICATION BRANCH, STATEMENT, AND PATH EXECUTION COVERAGE BRANCH AND MODULE TRACING AUTOMATED PROGRAM ANALYSIS DOCUMENTATION ASSERTION VIOLATIONS REACHING SET GENERATION
MENTEXT	MENTEL, INC. PALO ALTO, CA	OS	IBM/360/370	TEXT EDITING PROVIDES PROGRAM PREPARATION, TESTING AND DOCUMENT PREPARATION SERVICES FULL SCREEN EDITING CAPABILITIES
OPTIMIZER III	CAPEX CORP	COBOL	IBM 360/370	OPTIMIZES COBOL OBJECT CODE REMOVES DEAD CODE, REFORMATS SOURCE CODE PARAGRAPH TRACING AT ABEND ABEND REPORT ELIMINATES SYSUDUMP
PACE	TRW	FORTRAN	IBM 360 CDC 6400 UNIVAC 1108	STATEMENT TYPE STATISTICS STATEMENT INSTRUMENTATION
PANVALET	PANSOPHIC SYSTEMS OAK BROOK, IL	OS	IBM 360/370	BUILDS AND MAINTAINS LIBRARIES OF SOURCE PROGRAMS, JCL, DATA. CAN BE USED TO BUILD JOB STREAMS, TRANSFERS DATA SETS TO OTHER LIBRARIES-ON-LINE CAPABILITY

Table 4.1 (CONTINUED)

TOOL SYSTEM	DEVELOPER	LANGUAGE	COMPUTERS	CAPABILITIES OF TOOL
PET	MCDONNELL DOUGLAS	FORTRAN	CDC 6600/7600 IBM 360/370 HIS 6000 GE 600	STATEMENT TYPE STATISTICS STATEMENT INSTRUMENTATION
PROTEST	SYNEGETICS, CORP. BEDFORD, MA	COBOL	IBM 360/370	TEST DATA GENERATION
QUALIFIER	COMPUTER SOFTWARE ANALYSTS	FORTRAN COBOL JOVIAL ASSEMBLER		STATEMENT TYPE STATISTICS STATEMENT INSTRUMENTATION DATA INSTRUMENTATION
RXVP80	GENERAL RESEARCH CORP.	IFTRAN FORTRAN	CDC 6400/7600 IBM 370 VAX-11/780	AUTOMATED PROGRAM ANALYSIS DOCUMENTATION BRANCH EXECUTION COVERAGE CODING STANDARDS VIOLATIONS LOOP VIOLATIONS ASSERTION VIOLATIONS SET/USE VIOLATIONS PARAMETER VIOLATIONS REACHING SET GENERATION
SQLAB	GENERAL RESEARCH CORP.	FORTRAN IFTRAN JOVIAL J3-B PASCAL VPASCAL	CDC 6400/7600	DO LOOP VIOLATIONS SET/USE VIOLATIONS CODING STANDARDS VIOLATIONS PARAMETER VIOLATIONS BRANCH EXECUTION COVERAGE AUTOMATED PROGRAM ANALYSIS DOCUMENTATION SYMBOLIC EXECUTION FOR VERIFICATION ASSERTION VIOLATIONS REACHING SET GENERATION
STANDARDS AUDITOR	COMPUTER SOFTWARE ANALYSTS	FORTRAN COBOL COMPASS	CDC	CODING STANDARDS VIOLATIONS COMMON BLOCK VIOLATIONS DO LOOP VIOLATIONS
TSOBOL	SIGMATICS IRVINE, CA	COBOL	IBM 360/370	AUTOMATICALLY GENERATES COBOL PROGRAMS USING STATEMENTS STORED IN LIBRARIES. ALLOWS EDIT CAPABILITY
TSO SUPPORT AND STRUCTURED PROGRAMMING FACILITY	IBM	OS	IBM 370	FULL SCREEN CONTEXT EDITING WITH MULTIPLE-SCROLL AND SPLIT SCREEN CAPABILITY. KEEPS LIBRARY ACTIVITY STATISTICS. MAINTAINS PROGRAM LIBRARIES WITH A MENU-DRIVEN INTERFACE TO LANGUAGE COMPILERS.

Tools created to assist COBOL programmers are generally smaller software packages which address areas other than program verification. There are tools to automatically create the COBOL statements which are continually used in a COBOL program. This alleviates the tedious task of typing in the wordy areas of a program, which are usually the same in every COBOL program in that data processing department. When the code has been created, there are tools to automatically reformat the printed source code. The COBOL program, with the sections and paragraphs aligned and indented, is easier to read. There are tools to create flowchart pictures of the program logic, and tools to create cross-reference listings of data elements.

COBOL has been the language of business. The emphasis has been placed upon speed and efficiency of programming. Tools were developed to assist the programmer in the task of completing programs in a short time with the least amount of difficulties. Just recently, the need for quality assurance tools for COBOL programs has been recognized, and software verification tools to analyze COBOL code are beginning to be developed.

CAVS will organize the most important functions of several tools into one comprehensive software development aid; a unique package for the COBOL language. CAVS is designed to be a portable system requiring only one, easy-to-learn command language. Table 4.2 illustrates these advantages.

TABLE 4.2

CAVS: AN INTEGRATED TOOL

	CAVS	TEXT EDITORS	OPTIMIZER	ANALYZER/ DETECTOR	DCD II	IBM TSO
• PROGRAM REFORMATTING	X	X				
• STATIC ANALYSIS	X		X			
• INSTRUMENTATION	X			X		
• RUN-TIME ERROR ANALYSES	X			X		
• AUTOMATIC DOCUMENTATION	X				X	
• TEST ASSISTANCE	X					X
• STRUCTURED PROGRAMMING SUPPORT	X					

5 A SYSTEM FOR THE AUTOMATED VERIFICATION OF COBOL PROGRAMS

5.1 CAPABILITIES

The intent of Phase 1 was to study the most current techniques of program validation and testing, review available tools, develop new ideas and methodology, and design a COBOL AVS. With this information, coupled with our evaluation of the project scope (with respect to available resources: time, funding, and manpower), our outline of the capabilities to be supplied by the COBOL AVS is given in this section. A more detailed description is the CAVS Functional Description - Phase 1.

The AVS will include six major kinds of software development tools: (1) static analysis, (2) instrumentation, (3) testing analysis, (4) coverage assistance, (5) documentation and (6) reformatting. User interface will be through both batch and interactive terminals. The individual functions performed in each of these areas are described in Table 5.1.

A coded program is first submitted to the COBOL language compiler, which performs its syntactical functions. If errors occur, the programmer can make the corrections and resubmit the program to the compiler. The next phase is program verification using CAVS, illustrated in Fig. 5.1. Any of the Static Analysis, Documentation, and Testing Analysis functions, or Instrumentation, can be chosen by the programmer. Errors revealed by CAVS can be corrected at each step and the process can continue until the software is ready for production testing. The user retains control of the amount of testing coverage to be performed, choosing from a selection of CAVS functions. The figure illustrates the usual sequence of events to be followed in using CAVS.

TABLE 5.1.
SUMMARY OF CAVS FUNCTIONS

Functional Area	Command	Description	Default
Automatic	DOCUMENT	Macro Command, Requests Default set of Documentation and Cross-Reference Reports.*	No
	LIST	Lists COBOL source, formats and indents data and procedure division.	Yes
	CALLS	Cross references calling and called programs.	Yes
	FILES	Cross references program and file interaction.	Yes
	COPYTEXT	Cross references program and copy text interaction. Shows where copy texts are used within a system.	Yes
	LINKAGE	Cross references program vs. linkage section contents.	Yes
	IDENT	Cross references all identifiers in the system by program. Shows where defined, set, used.	No
	DATALOC	Cross reference of identifiers by their record position and program. Shows fields defined, set, used, even where identifier-names are different.	No

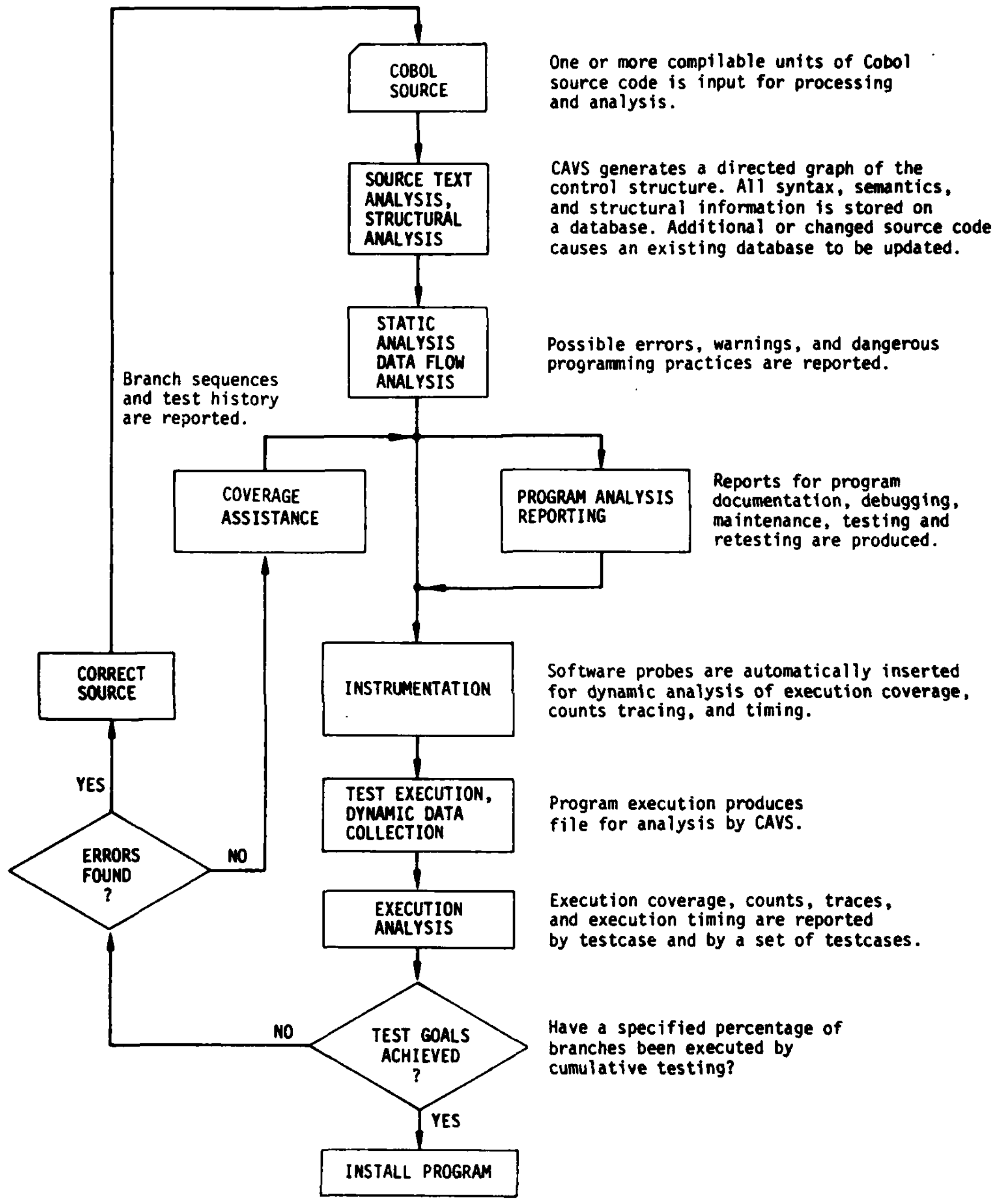
*MACRO commands are underlined. These commands cause all default commands for that functional area to be executed without having to request them explicitly.

TABLE 5.1. (CONTINUED)

Functional Area	Command	Description	Default
Automatic Documentation	PROFILE	Describe program interfaces, sizes, verb and I/O usage.	Yes
	INDEX	Index of reports generated and which programs are referenced in those reports.	Yes
Static Analysis	ANALYZE	Macro command, requests default set of static analysis reports.	No
	STATIC	General source analysis within a module, or within each module.	Yes
	CALLS	Analyzes interface between calling and called modules. Examines linkage sections, variables used, pictures of those variables.	Yes
	MOVGCORR	MOVE CORRESPONDING analysis. Checks corresponding data names for compatable pictures.	Yes
	REACHSET	Lists source statements on path between two statements in a program.	No

TABLE 5.1. (CONTINUED)

Functional Area	Command	Description	Default
Instrumentation	PROBE	Inserts diagnostic source statements which monitor execution control flow and time. Output from an instrumented program goes to an execution trace file.	No
Test Analysis and Assistance	TEST	Macro command, requests default test analysis reports of a specified execution trace file.	No
	COVERSINGL	Shows program source statements used during execution of single test run.	Yes
	COVERMULTI	Shows cumulative program source statements used during multiple test run.	
	EXTIME	Reports execution times of each invoked program unit.	Yes
	DETLTIME	Detailed timing analysis of program behavior. Reports total times by paragraph and section.	No
	NOTHIT	Shows program source, flags statements not hit by test case(s).	Yes
	SUMSINGL	Summarize single test case, showing numbers of paths not hit, percentages, execution summary.	Yes
	SUMALL	Summarize all test cases.	No



COBOL SOURCE
One or more compilable units of Cobol source code is input for processing and analysis.
SOURCE TEXT ANALYSIS, STRUCTURAL ANALYSIS
CAVS generates a directed graph of the control structure. All syntax, semantics, and structural information is stored on a database. Additional or changed source code causes an existing database to be updated.
STATIC ANALYSIS DATA FLOW ANALYSIS
Possible errors, warnings, and dangerous programming practices are reported.
Branch sequences and test history are reported.
COVERAGE ASSISTANCE
PROGRAM ANALYSIS REPORTING
Reports for program documentation, debugging, maintenance, testing and retesting are produced.
CORRECT SOURCE
INSTRUMENTATION
Software probes are automatically inserted for dynamic analysis of execution coverage, counts tracing, and timing.
YES
TEST EXECUTION, DYNAMIC DATA COLLECTION
Program execution produces file for analysis by CAVS.
ERRORS FOUND ?
NO
EXECUTION ANALYSIS
Execution coverage, counts, traces, and execution timing are reported by testcase and by a set of testcases.
NO
TEST GOALS ACHIEVED ?
Have a specified percentage of branches been executed by cumulative testing?
YES
INSTALL PROGRAM

AN-56536

Fig. 5.1. Overview of CAVS

Generally, static analyses are performed first. Reports are generated showing the results of those functions (outlined in Table 5.1). Once errors have been eliminated and the user is satisfied, the program can undergo dynamic testing analysis. Software probes are automatically inserted for dynamic analysis of execution coverage, for tracing execution sequences, for counting execution of segments, and for timing execution of segments. Program execution will produce a data collection trace file for analysis by CAVS, and listings are generated displaying the information gathered.

5.2 DESIGN

This section overviews the design particulars of CAVS. For a more comprehensive description, refer to the CAVS System/Subsystem Specification.

Once the needs of the user have been determined and the capabilities of the software package identified, the most important task is to create a design of the system. The design must demonstrate that the tool (1) will operate correctly and satisfy the user's requirements, (2) is written in language which is familiar and portable, (3) will execute with economic efficiency and within reasonable execution times, and (4) by its design, is easy to modify to keep up with change and is adaptable for enhancement.

The system's primary input is a collection of COBOL source text, which is recognized, parsed, and stored on the data base (composed of multilinked table structures). In this sense, it performs some of the functions of a compiler, but for the most part its purpose and operation are different. A compilable program is assumed, and therefore no syntax error checking is done. Unlike a compiler, the CAVS stores the source code in various representations (such as blank-delimited text form, statement token strings, and graphical representations). Attributes of the program (individual statements, parameters, symbols, etc.), will be saved in the data base and reconstructed in modified forms such as with testing coverage probes. The stored attributes are examined on a program-by-program basis or across program boundaries in order to evaluate the semantic consistency of the code or to generate summary and documentation information about the program.

THE DATA BASE

The CAVS data base comprises the collection of program data in a set of tables. The system is designed to handle large programs consisting of many subroutines, with the potential for run-to-run retention of data tables on auxiliary storage. The large data base is maintained in random access files called libraries, with each library holding a collection of tables. In core the working storage consists of allocated blocks of storage which contain active module data tables. Data transfers between the libraries and the working storage area, and between the working storage and analysis program, are controlled by the Library Manager system.

TABLE STRUCTURES

The tables that contain module information have a generalized structure. Access to table information is made through a section of the library manager called the access interface.

TOKENS

The functions of CAVS require manipulation of the COBOL source text and in many cases involve accessing specific elements of text such as variable names, keywords, operators, etc. Therefore CAVS stores text on its data base by breaking the text into its smallest meaningful elements (tokens).

COMMAND

The CAVS program is divided into a group of functional segments. Similar or sequential activities are combined in a segment and the activities and options are controlled by a set of segment commands. The first word of each command signifies which segment receives the command. Each segment contains a command recognition routine which processes each command sent to the segment to determine which options and activities are being requested.

STORAGE

The Nucleus or data base makes up the core-resident root of the system although to minimize storage requirements, some nucleus routines will be loaded into secondary storage until needed. Each of the function segments reside in secondary storage until called and loaded by the storage controller.

FUNCTIONAL SEGMENTS

The following is a brief description of each functional segment:

Command Decoding and Control: Process user input commands, output interactive response, and successively return each command to the overlay controller.

Initialization and Wrapup: Upon run initialization, open files, initiate execution of the storage manager, and set various global data; upon run termination, close files and (for batch mode) produce report index.

COBOL Source Text Analysis: Read COBOL 68/74 source and perform lexical scan, token recognition, symbol classification, and structural pointer construction.

Structural Analysis: Build program graph, store branches, and compute single-entry/single-exit reduction history used in data flow analysis.

Supplementary Table Building: Build tables needed for module dependence reporting and cross references.

Program Analysis Reporting: Produce selected reports at user command.

Instrumentation: Insert probes at program unit entries, exits, branches (depending upon type of instrumentation selected); define new testcase or end of all testcases.

Structural Testing Analysis: Analyze run-time execution trace file, produce coverage and trace reports, and update test history table.

Execution Timing Analysis: Analyze run-time execution trace and produce timing report.

Print Services: Print the contents of specified database tables.

DESIGN METHODOLOGY

We entertained the idea of a new system design for the COBOL AVS because outwardly COBOL appears different from the languages for which we have previously developed an AVS. Looking over the structure of the language as it is translated into the assembler level, we decided a new design would be unnecessarily expensive in time and effort. A more effective plan would be to translate the FORTRAN AVS code into logically equivalent COBOL, and then apply the necessary modifications so it processes COBOL and addresses the testing needs of COBOL programs. The available resources would be better utilized in improving our design, testing and debugging the AVS, and implementing more features.

After translation of the data base routines from Fortran to COBOL, the necessary Identification, Environment, and Data Division sections would be added. Most of the auxiliary, validation, and testing routines would be redesigned to operate on COBOL code.

In order to create a COBOL AVS which is portable, CAVS will be written in a subset of ANSI-COBOL 1974 which is compilable on the Univac, Honeywell, and DEC computers (Appendix B). Because these three machines have different system architectures, there are statements, specifications and formats unique to each machine which cannot be standardized. A variable front-end routine for each machine type will be created to deal with these few differences and allow the main body of CAVS to be identical for each system.

Figure 5.2 shows an overview of CAVS's design. It is similar to FAVS in that it utilizes a command translator, language recognizer, structure recognizer, and trace file decoder. COBOL tables will be used instead of FORTRAN arrays to store all data. The data base analysis components will be completely redesigned to analyze COBOL structures and test for COBOL errors.

5.3 OPERATION

CAVS is designed with the user in mind. The design of the user interface to the software tool is as important as the system/subsystem design. Often, a valuable software system sits on the shelf because the user cannot understand its operation, because the command language is too wordy, difficult or ambiguous, and because the user's guide is unreadable.

CAVS will operate in both batch and interactive modes. This increases portability of the system and allows each user to determine which is most convenient for his needs. The following examples give a brief description of the system's operation in the batch and interactive modes. For a more detailed description of the operational features of CAVS, refer to the Functional Description.

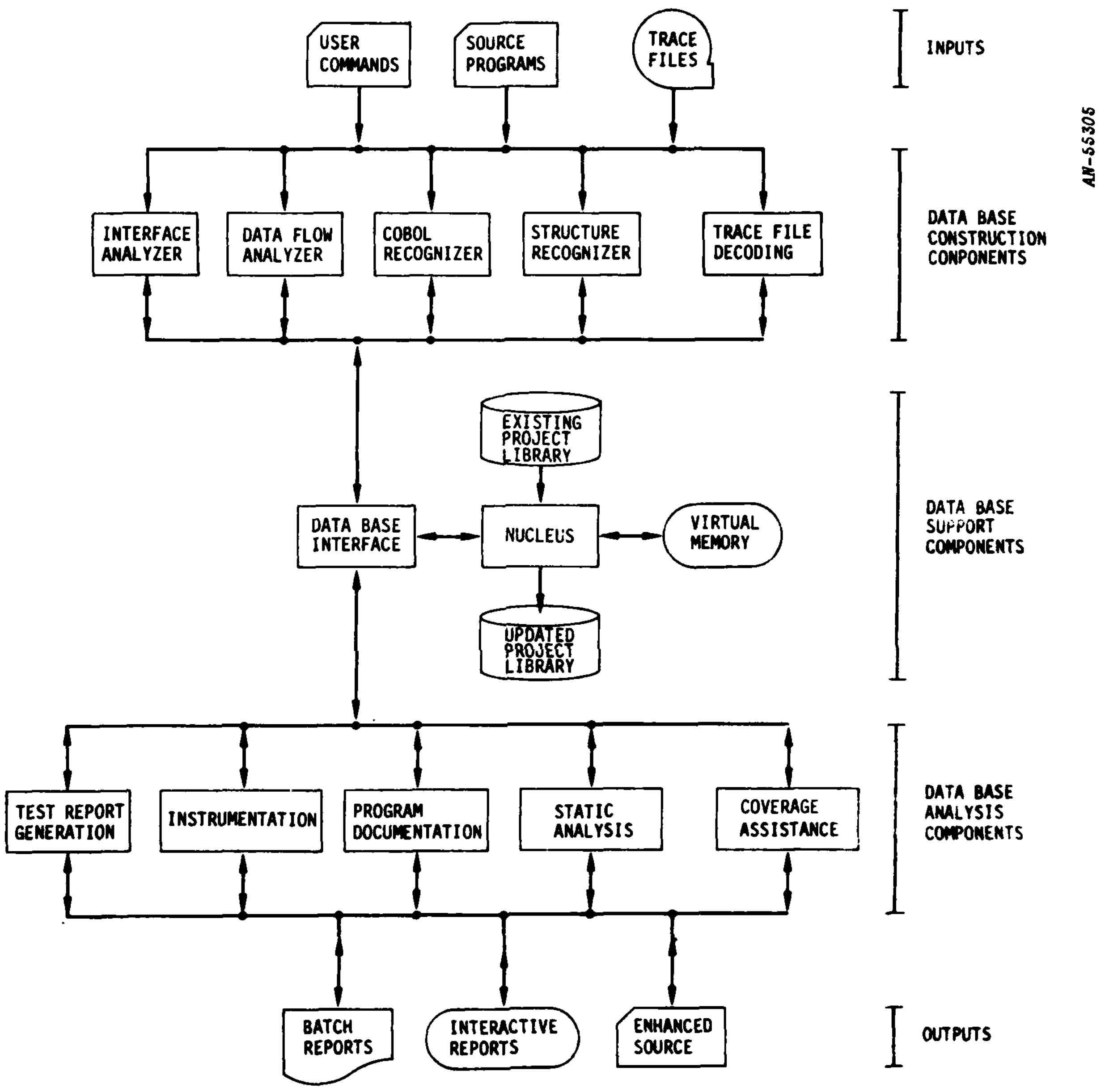


Fig. 5.2. CAVS Design Overview

BATCH OPERATION

An example of a batch runstream to perform Static Analysis, Documentation, and Testing Assistance for modules called MAINPROGRAM, SUB1, and SUB2 stored in file " DMA*PAYROLL. " is as follows:

```
@RUN GRCRUN,ACCTNO/passwd,10,100,5
@ASG,A DMA*PAYROLL.
@ASG,A CAVS*VERSION80.
@ASG,A DMA*TEMPFILE.
@ASG,A DMA*PAYROLL.OUTPUT1
@ASG,A CAVS*ECL.
@ADD CAVS*ECL.
OPEN INPUT.
FOR MODULES = MAINPROG SUB1 SUB2
ANALYZE STATIC CALLS
DOCUMENT LIST FILES IDENT
TEST UNIT TIMINGS COVERAGE
@EOF
@FIN
```

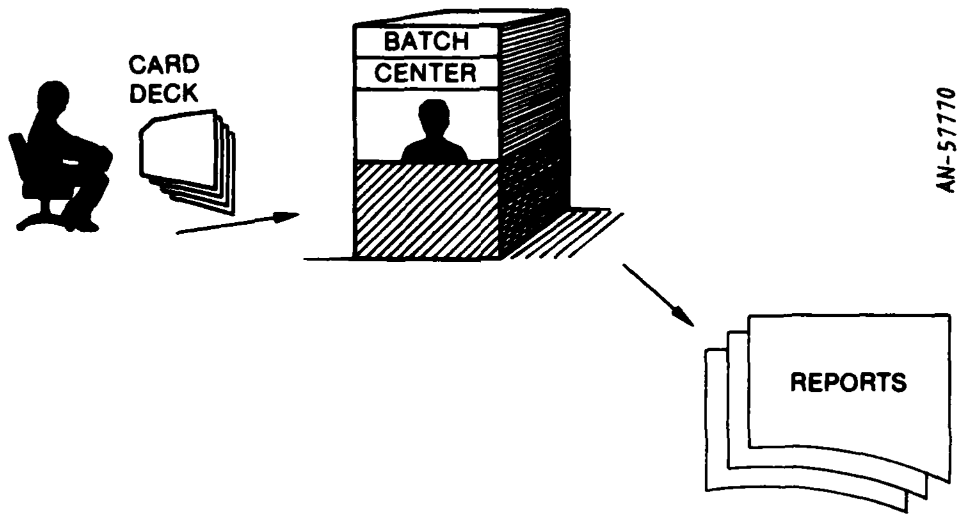
INTERACTIVE OPERATION

. CAVS interactive capability allows the user to create a job for execution, using interactive techniques, when use of the batch facility is not convenient. With the interactive system, the user responds to questions on a screen menu. The user will be able to create a CAVS job and execute the job while at the terminal.

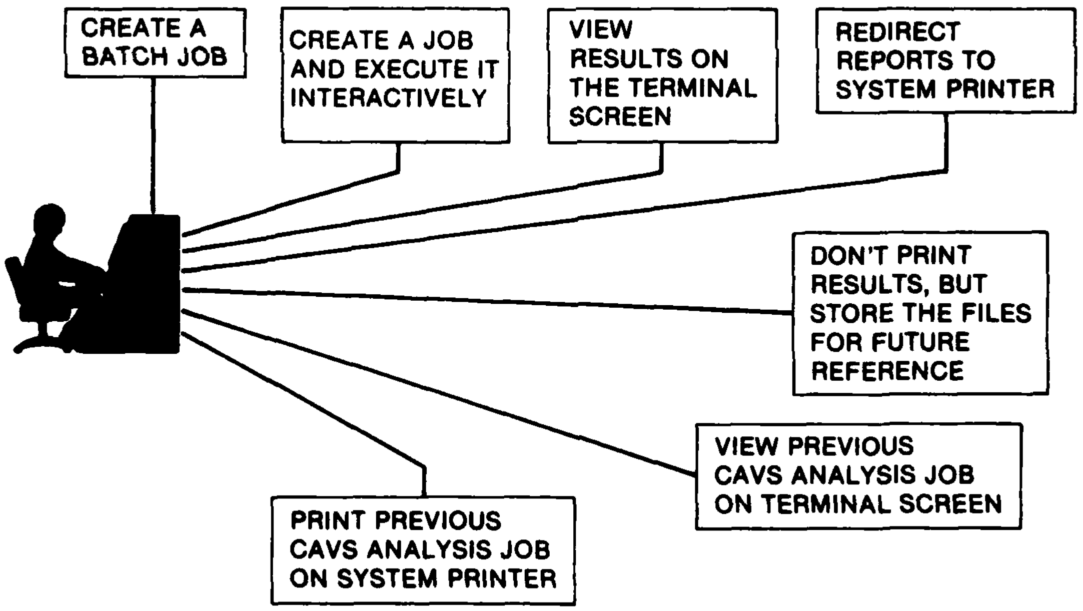
After sign-on, a main menu appears and the user enters the type of job processing he wishes to perform. Another menu, depending upon the response, appears automatically. The user can request CAVS analysis, browsing of previous CAVS reports, or printing of previous CAVS reports stored on the library. If CAVS analysis is requested, the appropriate menu appears and the programmer inputs the exact tailored functions desired. The user can, at any time, view the job runstream, or can request "help" information.

Before submitting jobs for execution, the programmer can direct the output to the terminal screen or the system line-printer.

Figure 5.3 illustrates some of the options available to CAVS batch and interactive users. Refer to the CAVS Functional Description for a more detailed explanation of the interactive processing capabilities.



(1) CAVS BATCH PROCESSING


CREATE A BATCH JOB
CREATE A JOB AND EXECUTE IT INTERACTIVELY
VIEW RESULTS ON THE TERMINAL SCREEN
REDIRECT REPORTS TO SYSTEM PRINTER
DON'T PRINT RESULTS, BUT STORE THE FILES FOR FUTURE REFERENCE
VIEW PREVIOUS CAVS ANALYSIS JOB ON TERMINAL SCREEN
PRINT PREVIOUS CAVS ANALYSIS JOB ON SYSTEM PRINTER

(2) CAVS INTERACTIVE PROCESSING

Figure 5.3. CAVS User Options

6 ENHANCEMENTS BEYOND CURRENT STATEMENT OF WORK

6.1 CONTINUING NEEDS OF CAVS USERS

We believe that every organization that builds or maintains software must be concerned with the need to improve productivity in software development and maintenance. RADC and DMA recognize this need. A study conducted by Planning Systems International, Incorporated, analyzed the DMA software development environments. The goal of the study was to develop a plan for establishing a "Modern Programming Environment" in each DMA installation.

Part of that plan recognized the importance of tools such as CAVS, and proposed methods of acquiring and installing additional tools, and supporting their effective use. The proposed enhancements in this chapter will make a significant contribution to the Modern Programming Environment at DMA and help to improve the management and productivity of the programming staff.

Table 6.1 summarizes the enhancements proposed for CAVS.

6.2 SUMMARY OF PROPOSED ENHANCEMENTS

Status Displays

The productivity of systems professionals and their management must be increased. One approach is to make more information about programs available to the staff. There are many tools that provide information, so much that programmers can be inundated with data. The problem is not so much one of not having information as one of not being able to find it. CAVS should be supplemented with a comprehensive on-line, interactive system which enables programmers to search, examine, and logically manipulate information about their software development activities. Managers should have access to status displays and reports revealing who uses CAVS and to what extent.

TABLE 6.1.

PROPOSED ENHANCEMENTS

Functional Area	Proposed Enhancements	Description
Demand Processing and Online Support	HISTORY	Show status of modules and entire CAVS library. Displays would show number of modules, percent modules documented, analyzed, tested, etc.
	Management Displays	Summarizes usage of CAVS. Shows number of modules and their test status.
	SUGGEST	Enables CAVS to suggest next course of action for each CAVS library entry. Serves as partner and bookkeeper during test process.
	Search-and-Select Capability	Enables CAVS user to browse output from multiple CAVS reports, combining and selecting data for specific programming problems.
	TUTORIAL	On-line documentation about CAVS functions and how to use them.
Program Restructuring and Modification	COBOL Restructuring	Reorganizes and simplifies program control structure. Reformats source for Eliminates GOTO's, puts redundant code in performed paragraphs.
	Control Flow Picture	Shows coherent, separately performed segments of unstructured COBOL. Simplifies preparing programs for automatic restructuring.
	Program Structure Charting	Draws hierarchical VTOC* charts for structured, GOTO-free COBOL programs. Provides graphic programs. Could automatically generate charts required by installation standards.
	Coding Practices Analyzer	Flags poor practices and usage of error-prone constructs. Such constructs as ALTER and GOTO's would be flagged and counted.

TABLE 6.1. (CONTINUED)

Functional Area	Proposed Enhancement	Description
Automatic Documentation	INTERFACE	Summarizes interfaces with other modules. Describes Documentation linkage section, files, file formats, call statements, parameters, external switches.
	LAYOUT	Automatic generation of record layouts for FD's and for working storage areas flagged by user or used in read into, write from operations.
	SPECIAL	List special names, system-dependent and non-standard operations, cross-reference features, operating formats which may hinder conversions, upgrades, etc.
Static Analysis	ANALYZE	Expand analyses to test for simple infinite loops
	SEGMENT	Analyze use of segmentation feature. Evaluate control and usage of overlays. Assist in debugging programs requiring overlays.
Program Development	COBOL ASSERTIONS	Produces execution-time diagnostics and error messages about program behavior.
COBOL Dialects	Recognize WWMCCS, Old COBOLS, COBOL-F	Extend recognition, documentation and analysis functions of CAVS to process additional dialects of COBOL. • WWMCCS - Military Dialect for Command, Control, Operating Systems • OLD COBOLS - Process older dialects with keywords eliminated from COBOL-68, COBOL-74 Standards.

*Visual Table of Contents

Tutorial

Training of new employees is expensive and in times of high turnover it can be almost impossible. Many installations recruit people from user departments for positions in the systems development area. While this process provides employees with firsthand knowledge of user departments, it also means more training is required to produce a good systems professional. As part of the comprehensive on-line system, we propose that CAVS be equipped with a tutorial component which would guide novice users and supplement the knowledge of experienced users.

Automatic Restructuring and Charting

Preserving an investment in old software while introducing modern programming techniques is very difficult. As more experienced programmers are promoted or leave an installation, the number of people who understand and can quickly fix old, GOTO-ridden programs declines. Automatic restructuring of old programs would enable CAVS users to improve the readability and maintainability of their programs, while preserving the integrity of their systems. This process could be supplemented by automatic charting programs to produce program structure charts and visual tables of contents (VTOCs). These VTOCs document the overall control structure of GOTO-free programs. These tools could reduce the cost of maintaining, improving, and documenting COBOL systems.

Conversion Support

Upgrading an installation's computer or operating system is frequently a painful task. While improvements in machine efficiency or operating system capabilities are the eventual results, converting old programs is time-consuming and error-prone. Economic benefits to be gained from using a data base management system or a more modern compiler must be weighed against the cost of conversion. By enabling CAVS to recognize old dialects of COBOL, Government installations could use CAVS documentation, restructuring, and static analysis features to upgrade large volumes of old, non-standard COBOL programs.

COBOL Assertions

Assertions are statements which are added either manually or automatically to monitor the execution-time behavior of a program. Their most valuable function is the ability to state what the expected behavior of a program should be, and then issue diagnostics when an error condition is detected. They are distinct from regular logical and input-output statements in that their operation can be easily switched on or off.

CAVS assertions would be coded with COBOL-like statements, and translated by CAVS into valid COBOL.

WWMCCS COBOL

CAVS can be modified to recognize other specialized dialects of COBOL such as WWMCCS. WWMCCS is implemented on Honeywell computers, with an operating system written in a language very much like COBOL. A WWMCCS implementation of CAVS would be able to analyze the WWMCCS operating system and applications that run on it. Automated tools for testing and increasing the reliability of this system should be developed.

Interaction of CAVS with FAVS

FAVS, the FORTRAN Automated Verification System already installed at RADC and DMA, performs most of the functions for FORTRAN and DMATRAN programs that CAVS will do for COBOL. By enabling CAVS and FAVS to talk to each other, systems which use both of these languages could be verified in a more comprehensive and thorough manner.

6.3 CHARACTERISTICS OF PROPOSED ENHANCEMENTS

These enhancements are grouped by functional areas similar to the existing functions of CAVS. A summary of the enhancements is shown in Table 6.1. All features would be implemented for on-line users; control would be by means of menus. For automatic restructuring of large programs, and for program graph analysis, the job control language could be produced on-line, but actual execution should take place in batch mode. Users could print text from the tutorial, but on-line access to the same information would result in great savings in programmer time. Allowing users to examine CAVS output at the terminal and print only what was necessary would also reduce the volume of paper produced by CAVS.

6.3.1 Demand Processing and On-line Support

CAVS on-line users will be able to build batch jobs or to execute all CAVS functions on-line, in "demand mode." While this approach provides users with access to the most commonly needed programming information, the problem of how to help the programmer use and manage that information should be addressed. CAVS can be made intelligent enough to:

- Track the progress of modules through the system
- Report their status to users and management
- Suggest courses of action
- Explain what happens when a particular course is selected
- Search and manipulate CAVS report output in ways tailored to a specific user-defined problem.

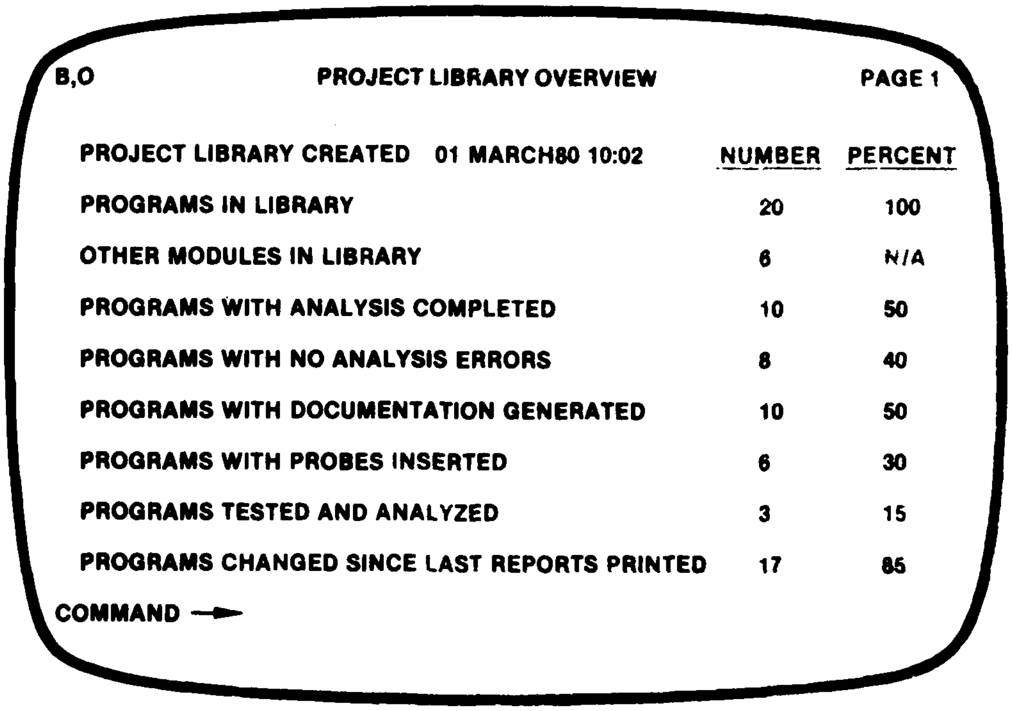
All of these functions should be available to on-line users.

HISTORY is a proposed system-wide data logging function. It would record:

- When modules were added to a library
- How many versions of a module had been added
- If and when a module was analyzed, documented, instrumented, or tested

- Which users and projects were actually using CAVS, and to what degree.

Status displays for managers and programmers would reveal the current state of program development for an entire project or for a single program. How the tool was actually used could then be more accurately determined. CAVS should collect enough data about its own behavior to guide managers and programmers in the most effective use of the tool. Figure 6.1 illustrates a proposed project library overview display, showing, in CAVS terms, the number of modules at each stage of program development. Their percentage of the total modules in the project library would also be shown. Information about how and when CAVS had processed a single module would be shown on a display like Figure 6.2.

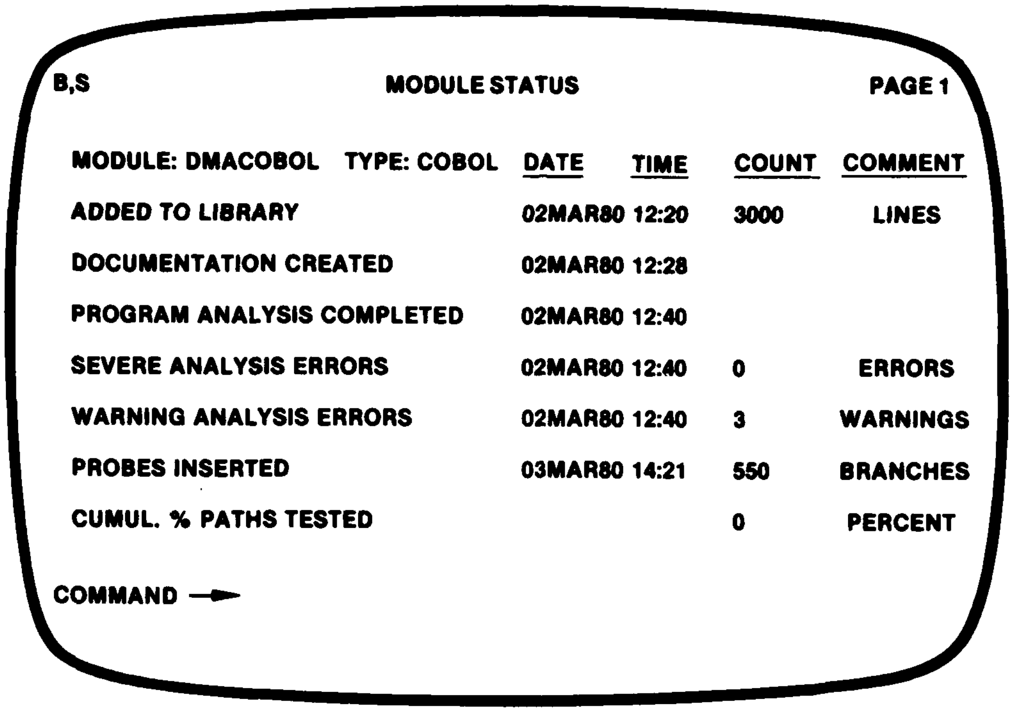


B,O PROJECT LIBRARY OVERVIEW PAGE 1

PROJECT LIBRARY CREATED 01 MARCH80 10:02	NUMBER	PERCENT
PROGRAMS IN LIBRARY	20	100
OTHER MODULES IN LIBRARY	6	N/A
PROGRAMS WITH ANALYSIS COMPLETED	10	50
PROGRAMS WITH NO ANALYSIS ERRORS	8	40
PROGRAMS WITH DOCUMENTATION GENERATED	10	50
PROGRAMS WITH PROBES INSERTED	6	30
PROGRAMS TESTED AND ANALYZED	3	15
PROGRAMS CHANGED SINCE LAST REPORTS PRINTED	17	85

COMMAND →

Figure 6.1. Project Library Overview Display

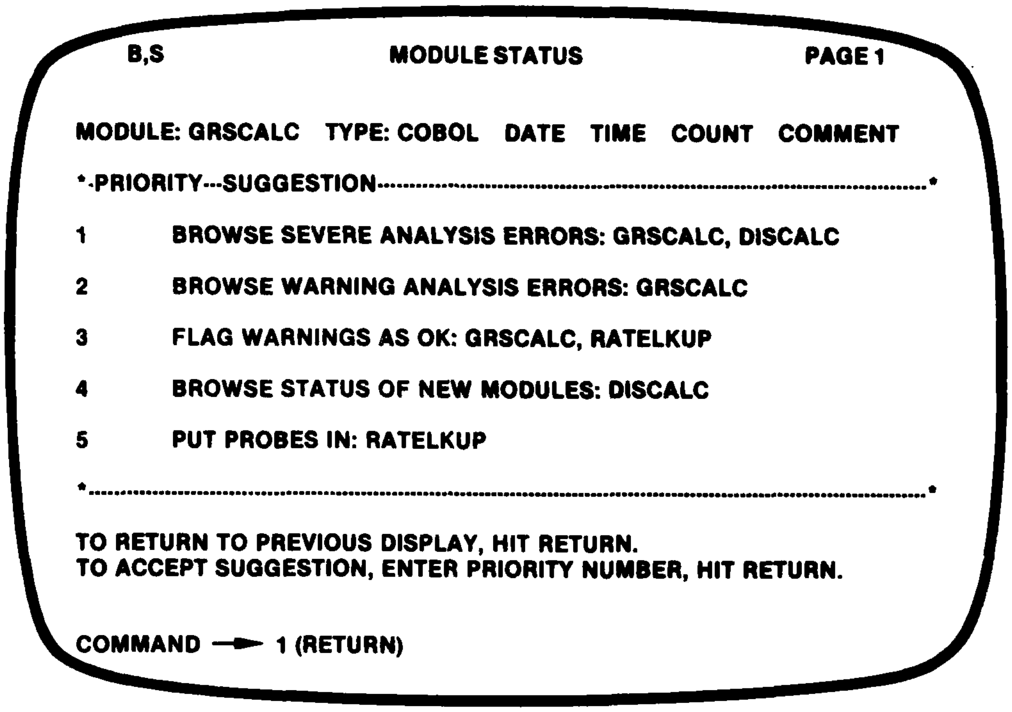


```
B,S                          MODULE STATUS                          PAGE 1

  MODULE: DMACOBOL   TYPE: COBOL   DATE     TIME    COUNT   COMMENT
  ADDED TO LIBRARY                 02MAR80 12:20    3000    LINES
  DOCUMENTATION CREATED            02MAR80 12:28
  PROGRAM ANALYSIS COMPLETED       02MAR80 12:40
  SEVERE ANALYSIS ERRORS           02MAR80 12:40    0       ERRORS
  WARNING ANALYSIS ERRORS          02MAR80 12:40    3       WARNINGS
  PROBES INSERTED                  03MAR80 14:21    550     BRANCHES
  CUMUL. % PATHS TESTED                             0       PERCENT

COMMAND →
```

Figure 6.2. Status Display for One Module

Suggestions and Guidance

CAVS could use its HISTORY information to suggest courses of action to on-line users. On-line users are already guided through a work cycle. Program development also follows a predictable cycle of coding, analyzing, testing, and documentation. CAVS could look at a module's status and propose what actions should be done next. In Fig. 6.3, for example, CAVS has been asked to suggest actions for GRSCALC, a module which is already the object of a status display. After checking HISTORY for GRSCALC and for the project library, CAVS suggests reviewing errors found in GRSCALC. In this example, alternatives based on what is known about other modules (in the project library) are also presented.



```
B,S                       MODULE STATUS                    PAGE 1

MODULE: GRSCALC   TYPE: COBOL   DATE   TIME   COUNT   COMMENT

*-PRIORITY---SUGGESTION-------------------------------------------*

1        BROWSE SEVERE ANALYSIS ERRORS: GRSCALC, DISCALC

2        BROWSE WARNING ANALYSIS ERRORS: GRSCALC

3        FLAG WARNINGS AS OK: GRSCALC, RATELKUP

4        BROWSE STATUS OF NEW MODULES: DISCALC

5        PUT PROBES IN: RATELKUP

*-----------------------------------------------------------------*

TO RETURN TO PREVIOUS DISPLAY, HIT RETURN.
TO ACCEPT SUGGESTION, ENTER PRIORITY NUMBER, HIT RETURN.

COMMAND ──► 1 (RETURN)
```

Figure 6.3. CAVS Suggestion Capability

SUGGEST and HISTORY would relieve experienced programmers of much of the bookkeeping done during development and maintenance. Inexperienced programmers and new CAVS users could become productive more quickly. If the suggestions were accepted, CAVS could begin executing the function immediately or generate a job to execute the function in batch mode.

CAVS programmers must now do at least three things during the course of solving a problem: determine what reports contain data they need, produce the reports, and extract the necessary information. SUGGEST and TUTORIAL capabilities can help in the first two activities. Adding special logic capabilities to CAVS would make the last task much easier. We propose equipping CAVS with programs which scan its own data base and recognize commonly-used COBOL constructs and user-created data items. This would not be a text searching function, something that could be done with a text editor, but a search based on the logical properties of COBOL programs. The LOGIC processor could then:

- Extract data sets of data names, file information, and COBOL verbs.
- Sort and merge extracted data sets in ways specified by the programmer.
- Create new data sets which represent the logical intersection of two or more CAVS reports. For example, a list of all CALLs and a list of identifiers could be merged to produce a list of all CALLs which reference those identifiers.
- Save and delete the data sets created by the LOGIC processor.

6.3.2 Automatic Restructuring

GRC's Fortran Automatic Verification System contains a function for automatically restructuring FORTRAN programs to eliminate GOTOs. There are several problems with the restructuring procedure as it now exists:

- Automatic restructuring without additional human input produces an equally bad program without GOTOs.
- Restructuring a program causes documentation of the old program's internal logic to become obsolete.
- Restructuring large programs is time-consuming and expensive.

CAVS should provide restructuring for COBOL programs, but (based on the experience of users at DMA and GRC) we feel an improved, cost-effective restructuring module can be built. It should provide these services:

- Automatic analysis of existing program structure.
- Easy-to-read program graph reports. These would show how to break a large program into discrete, easily understood (by people) and easily structured (by computer) chunks.
- Error and diagnostic reports. Any COBOL constructs that render the program unstructurable would be reported.
- Automatic restructuring of COBOL source programs. CAVS would be able to restructure any COBOL dialect it could recognize.
- Automatic chart generation. Documentation (VTOCs and/or flowcharts) for newly structured programs could be generated mechanically. Cost of documenting these new versions of well-tested programs would be reduced.

6.3.3 Automatic Documentation

CAVS is designed to be portable across computer models and operating systems. Its automatic documentation modules concentrate on the portable portions of COBOL programs. But information about non-portable language constructs, file structures, and program interfaces can also be important. During conversion from one computer to another or one operating system to another, knowledge of these non-portable system characteristics is vital. Conversions can be simplified and expedited when this information is extracted and presented automatically.

INTERFACE is a proposed report that summarizes the interface characteristics of a COBOL program. It would identify the following, on one page if possible:

- File name and external file code

- File organization and format
- CALLs to other modules
- Information received from and passed to the operating system
- Linkage section variables and their attributes
- Presence or absence of system-dependent constructs, e.g., DECLARATIVES, FILE-STATUS, etc.

INTERFACE would conveniently organize and present how COBOL programs communicate with the host operating system and with each other.

LAYOUT is a function requested by the staff at DMA. This option would create record layouts from COBOL file descriptions (FDs) and from Working Storage areas. It could be extremely useful when working with utilities which reference data items by location instead of variable-name. LAYOUT would determine the size and physical location of each field in a record from the field's *COBOL picture.* For each file and related Working Storage area, LAYOUT would show:

- File name and external code
- File organization and format
- Field name position
- Field characteristics
- Any overlapping or redefined identifiers
- Where used in the system (optional)

SPECIAL is a proposed function which could audit programs for non-portable and non-standard language constructs. Unlike the FIPS flagger, which is available for some compilers, SPECIAL could be tailored to an installation's needs. While INTERFACE provides a high-level summary of interface characteristics, SPECIAL would flag these items in the source. SPECIAL would be of great value in monitoring the work of contractors and new programmers who do not adhere to or do not know a shop's coding standards.

Upgrading of programs written in pre-1968 COBOL dialects could be simplified through the use of CAVS. Program profiles, the proposed INTERFACE and SPECIAL reports, and the restructuring function could be used to upgrade and convert old programs.

6.3.4 Static Analysis

Research on methods of static analysis will continue throughout the CAVS development. Infinite loops are a common programming problem; they waste computer time and are a common error of novice programmers. CAVS now has some capabilities for discovering graphical loops; additional research may enable CAVS to detect more complex, logically infinite loops.

Neither the Univac 1100/80 series nor the Honeywell 6000 series provides virtual memory. In order to execute large COBOL programs, they use a segmentation feature to break programs up and execute them in pieces to reduce the amount of memory required. SEGMENT is a proposed CAVS feature which would analyze a program's control structure and identify sections that are good candidates for segmentation. This feature could be used in conjunction with the Navy's segmentation tool to improve the performance of large programs on machines without virtual memory.

6.3.5 COBOL Assertions

Assertions are logical expressions which may be either true or false, but are expected to be true. If these expressions are not true, the condition is known as an assertion violation. Assertions are used primarily for two purposes: to state formally what a program should do (or not do), and to monitor the program's behavior during execution. When used consistently and efficiently, assertions can simplify the testing and debugging of programs.

CAVS users should have the option to put assertions in their source code, and then have CAVS translate them into valid COBOL. CAVS would support logical assertions (which evaluate a user-defined logical expression and take action when that expression is false) and also support input-output assertions (which monitor values read into, written from, or passed to a program).

Most computer manufacturers provide some assertion-like capabilities with their COBOLs. But these statements are part of the source program and are not easily removed. CAVS assertions, on the other hand, may be switched on or off at compile time.

CAVS users would be provided with a compile procedure which first passes their source to the CAVS assertion processor. At that time, they will have the option to:

- Switch the assertions on, by directing CAVS to translate them into valid COBOL
- Switch the assertions off, and have CAVS mark them as comments so that they remain in the source code as documentation
- Switch the assertions off, and remove them from the source code

6.3.6 WWMCCS COBOL

The Government has invested over nine years and significant resources in the WWMCCS* operating system and resident software. A special dialect of COBOL was also maintained for WWMCCS, which is implemented on Honeywell 6000 series computers. CAVS should be modified to provide WWMCCS programmers with tools to analyze, document, and test their programs more effectively. Specifically CAVS would be modified to:

* World Wide Military Command and Control System

- Run on WWMCCS architecture and operating systems
- Recognize WWMCCS COBOL
- Produce all CAVS reports and displays in formats suitable for WWMCCS hardware

6.3.7 CAVS and FAVS Compatibility

DMAAC and DMAHTC already have a GRC-developed tool for FORTRAN users, FAVS. FAVS and CAVS will perform many of the same functions, but each tool is restricted to dialects of one language. Each tool provides a permanent data base which contains descriptions of source text and testing history. (In CAVS terminology, this is the project library; in FAVS terminology, it is a restart file.) GRC will implement CAVS, and modify FAVS, so that:

- FAVS can process and update a CAVS project library.
- CAVS can process and update a FAVS restart file.
- FAVS can process trace files written by instrumented COBOL programs.
- CAVS can process trace files written by instrumented FORTRAN programs.

CAVS and FAVS can then be used to analyze, test, and document programs which contain both COBOL and FORTRAN modules.

A system containing both COBOL and FORTRAN modules would be verified as follows:

- CAVS would statically analyze, document, instrument, and reformat COBOL modules and add information about them to a project library (restart file).
- FAVS would statically analyze, document, instrument, and restructure FORTRAN modules and add information about them to a restart file (project library).

- Either CAVS or FAVS would be used to produce test coverage reports resulting from execution of instrumented modules.
- Either CAVS or FAVS would be used to update the testing history information on the project library (restart file).
- Either CAVS or FAVS would be used to obtain source text listings and system-wide documentation regarding entry points, externals, global variables, and file usage.

No significant changes are required to the CAVS Functional Description or CAVS System/Subsystem Specification to accommodate this compatibility. On the other hand, FAVS will require significant additional effort in order to upgrade the FAVS restart file, instrumentation processing, and test coverage analysis. In order to insure trace file compatibility between CAVS and FAVS, only full word integer data will be written to trace files by instrumented routines. Since the CAVS project library and FAVS restart file must contain at least some character data in addition to full word integer data, it may be more difficult to achieve compatibility in this area. If character data written by a COBOL program cannot be read by a FORTRAN program (or vice versa), it may be necessary to use both COBOL or both FORTRAN routines to read and write the project library in CAVS and the restart file in FAVS.

The following features will be added or enhanced in FAVS.

- Parameter checking for calls to COBOL routines.
- Relative organization for the FAVS restart file (it currently is a sequential file in FAVS and a random file in CAVS).
- Updating in place of a restart file or creation of a new copy (similar to CAVS).

- Inclusion of source text, structural information, and testing history on the restart file (for compatibility with CAVS).
- More flexible and efficient instrumentation of FORTRAN source (as well as compatibility with COBOL trace files).
- Improved test coverage and test history reports (oriented to the user's original source code as in CAVS).
- A menu-driven interactive interface for compatibility with CAVS use.

Parameter checking for calls to FORTRAN routines will be added to CAVS.

7 BIBLIOGRAPHY

Alberts, D., "The Economics of Software Quality Assurance", Proceedings of COMPSAC 77, Computer Software and Applications Conference, November 1977, p. 222.

Andrews, D. M., Benson, J. P., Advanced Software Quality Assurance, Software Quality Laboratory User's Manual, General Research Corporation, CR-4-770, May 1978.

Benson, J. P., et. al., Software Verification: A State-of-the-Art Report, GRC, CR-1-638, March 1978.

Boyer, R. S., Elspas, B., Levitt, K. N., Select--A System for Testing and Debugging Programs by Symbolic Execution," Submitted to the 1975 International Conference on Reliable Software, April 1975.

Brooks, N. B., Gannon, C., JAVS, Jovial Automated Verification System, Vol. 3, General Research Corporation, CR-1-722, December 1976.

Brooks, N. B., Gannon, C., JAVS Jovial Automated Verification System, Vol. 2, General Research Corporation, CR-1-722/1, June 1978.

Brown, J. R., Lipnow, M., "Testing for Software Reliability, Proceedings of COMPSAC 77 Computer Software and Applications Conference, November 1977, p. 21.

Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs" IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976.

Fischer, K. F., "Software Quality Assurance Tools: Recent Experience and Future Requirements," Software Quality and Assurance Workshop, San Diego, November 1978.

Gerhart, S., Yelowitz, L., "Observations of Fallibility in Applications of Modern Programming Methodologies", Proceedings of COMPSAC 77 Computer Software and Applications Conference, November 1977, p. 86.

Glass, R. L., Real Time Software Debugging and Testing: Introduction and Summary, The Boeing Company, September 1979.

Holden, M. T., "Semi-Automatic Documentation of B-1 Avionics Flight Software Global Data," Naecon 1978 Record.

Howden, W. E. "Effectiveness of Software Validation Methods," Infotecn: Software Testing, Vol. 2, 1979.

Howden, W. E., "Reliability of the Path Analysis Testing Strategy", Proceedings of COMPSAC 77 Computer Software and Applications Conference, November 1977, p. 99.

Howden, W. E., "An Evaluation of the Effectiveness of Symbolic Testing," Software - Practice and Experience, Vol. 8, 1978.

Howden, W. E., "Theoretical and Empirical Studies of Program Testing," IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978.

King, J., "Symbolic Execution and Program Testing", Proceedings of COMPSAC 77 Computer Software and Applications Conference, November 1977, p. 191.

Miller, E. F., Jr., Methodology for Comprehensive Software Testing, General Research Corporation, CR-1-465, February 1975.

Miller, E. F., Jr., Paige, M., Bendon, J., Wisehart, W., "Structural Techniques of Program Validation", Proceedings of COMPSAC 77 Computer Software Applications Conference, November 1977, p. 179.

Miller, E. F., Jr., "Toward Automated Software Testing: Problems and Payoffs", Proceedings of COMPSAC 77 Computer Software and Applications Conference, November 1977, p. 16.

Moriconi, M. S., A System for Incrementally Designing and Verifying Programs, Vol. 1, USC/Information Sciences Institute, November 1977.

Ramamoorthy, C. V., Ho, S.F., "Testing Large Software with Automated Software Evaluation System", Proceedings of COMPSAC 77 Computer Software and Applications Conference, November 1977, p. 121.

Stucki, L. G., et al, Software Automated Verification System Study, McDonnell Douglas Astronautics Company, January 1974.

"SURVAYOR, The Set-Use of Routine Variables Analysis Program," TRW Brochure, 1975.

APPENDIX A

MATERIAL REFERENCED IN PHASE I

The following documents were referenced in Phase 1 or will be used in the coding implementation of the AVS in Phase 2.

Documents Describing the Implementation Environments

	TITLE	VENDOR'S NUMBER
1.	VAX/VMS Command Language User's Guide	AA-D023A-TE
2.	VAX-11 COBOL-74 Language Reference Manual	AA-C985A-TE
3.	VAX-11 COBOL-74 User's Guide	AA-C986A-TE
4.	VAX/VMS RMS Record Management Services	AA-D024B-TE
5.	Honeywell General Comprehensive Operating Supervisor (GCOS)	DD19D
6.	Honeywell Control Cards Reference Manual (GCOS)	DD31C
7.	Honeywell Standard COBOL-68 Reference Manual	DE17
8.	Honeywell Standard COBOL-68 User's Guide	DE18
9.	Honeywell Standard COBOL-74 Reference Manual	DE01
10.	Honeywell Standard COBOL-74 User's Guide	DE02
11.	Sperry Univac 1100 Series Operating System Programmer Reference	UP-4144

12.	Sperry Univac 1100 Series ANSI Standard COBOL (ASCII) - 1974	UP-8582
13.	Sperry Univac 1100 Series COBOL (ASCII) Supplementary Reference	UP-8584

Documents Referenced During the Research Phase

TITLE		REFERENCE/ REPORT NUMBER
1.	JOVIAL Automated Verification System JAVS Technical Report Reference Manual	RADC TR-77-126
2.	FORTRAN Au*omated Verification System FAVS User's Manual	RADC TR-78-268
3.	COBOL Instrumentation Packaage (CIP) For the Honeywell 6000	NARDAC 88-5002B TN-02
4.	COBOL Instrumentation Package (CIP) For the Univac 1108	NARDAC 88-50002B TN-01
5.	JAVS Technical Report, Volume 3 Methodology Report	GRC CR-1-722
6.	JOVIAL J73 AVS Part I - Technical Proposal	GRC DP-9874
7.	SID - System for Incrementally Designing and Verifying Programs	USC ISI/RR-77-65
8.	COBOL Segmentation Analysis Package for the Univac 1100 Series Computer	NARDAC 91Z7064 TN-03

Documents Referenced During the Research Phase (continued)

	TITLE	REFERENCE/ REPORT NUMBER
9.	Static Profile and Dynamic Behavior of COBOL Programs SIGPLAN Notices Vol. 13, No. 1, April 1978	
10.	A Model for Estimating the Number of Residual Errors in COBOL Programs. Cecil E. Martin, Ph.D. Thesis.	AFIT CI-77-97
11.	The Verification of COBOL Programs U.S. Army Computer Systems Command Technical Documentary Report L. Robinson, M.W., Green, J. M. Spitzen.	SRI 3967
12.	DATAPRO Directory of Software DATAPRO Research Corporation 1805 Underwood Blvd., Delran, N.J.	
13.	CAPEX Optimizer III: Analyzer User Guide Capex Corporation, Phoenix, AZ	S02-1077-135-(03-0979)
14.	CAPEX Optimizer III: Detector User Guide Capex Corporation, Phoenix, AZ	S02-1077-136-(03-0979)
15.	COBOL Optimization Techniques Capex Corporation, Phoenix, AZ	SMG-0676-39(03-0379)
16.	Structured ANS COBOL, Part 1 Paul Noll Mike Murach and Associates, Inc., Fresno, CA	
17.	Structured ANS COBOL, Part 2 An Advanced Course Paul Noll Mike Murach and Associates, Inc., Fresno, CA	

Documents Referenced During the Research Phase (continued)

TITLE	REFERENCE/ REPORT NUMBER
18. MRI System 2000/80 Procedure Language Feature - COBOL MRI Systems Corporation	
19. DMA Programming Support Library Structured COBOL Precompiler, Vol. 3	
20. Advanced Software Quality Assurance Software Quality Laboratory User's Manual	GRC CR-4-770
21. Data Correlation and Documentation System (DCD) CGA Computer Associates Rockville, Maryland	
22. Data Administration Systems (DAS) CGA Computer Associates Rockville, Maryland	

APPENDIX B

COBOL IMPLEMENTATION SUBSET

This Appendix discusses the subset of COBOL to be employed in the coding, testing, installation, and maintenance of the COBOL Automated Verification System. CAVS is to be installed on the Honeywell Series 6000 at Rome Air Developent Center, and on the Univac 1100 Series machines of the Defense Mapping Agency. It will first be coded and tested on the Digital Equipment Corporation VAX 11/780 at General Research Corporation. All machines support full ANS COBOL-1974 ASCII compilers, and CAVS will use them.

In each computer's user manuals there is a method of identifying those language constructs which are extensions of the ANSI COBOL-74 standard. CAVS is designed not to require these computer-specific language extensions. If, however, using an extension would significantly improve the performance or utility of CAVS on a particular machine, it will be used. GRC will first request the approval of the contracting agency. Any such code will be isolated in subprograms wherever possible.

There is one major exception to the above restriction. CAVS is based on a system originally written in FORTRAN, that makes extensive use of FORTRAN COMMON blocks. CAVS execution speed would be severely reduced if an equivalent construct were not available for COBOL. Since CAVS will function primarily as an on-line program, deviation from the ANS Standard COBOL is justified.

VAX-11 COBOL supports a construct like COMMON, called EXTERNAL. The Honeywell Series 6000 Standard COBOL supports a similar construct, called LABELED COMMON. The Univac 1100 Series COBOL supports a COMMON-STORAGE SECTION also, although its properties and syntax differ from those of both VAX and Honeywell implementations.

This Appendix will use a format similar to the Univac ASCII COBOL Programmer's Reference. The implementation subset will be described in numbered sections, in the following order:

1. General concepts
2. Major omissions from ANSI Standard COBOL
3. Identification division
4. Environment division
5. Data division
6. Procedure division
7. Table handling
8. General I/O considerations
9. Sequential I/O
10. Relative I/O
11. Library
12. Debug
13. Inter-Program Communication

Paragraphs within these standards will be numbered to simplify reference to a specific standard.

B.1 GENERAL CONCEPTS

1. CAVS will use the ASCII compilers for all three implementations. Characters stored on disk, and characters used in communication between calling and called programs, will be in ASCII format. Numeric data will be almost entirely full-word binary integers. Numbers used for reporting, diagnostic, and display purposes will be moved to the appropriate DISPLAY-usage ASCII fields.

2. Comments take three forms in COBOL:

   - An asterisk in column 7 is the standard comment character.
   - A "D" in column 7 can be either a comment or an indicator of debug statements.

- Certain keywords delimit an entire paragraph of comments.

CAVS will use the asterisk and the "D" to define comments. DATE-COMPILED will be used to insert the compile date in the source listing.

No other methods of delimiting comments, such as "REMARKS" in the Identification Division and "NOTE" in the Procedure Division, will be used. These have been removed from the 1974 standard.

3. CAVS must analyze itself during the self-test portion of its installation. This capability will also be used to scan for non-portable language constructs which would prevent successful conversion.

B.2 OMISSIONS FROM ANS COBOL-74

The following major COBOL features will not be used in the final, installed versions of CAVS:

1. Control Division
2. Sort/Merge
3. Report Writer
4. Communications Division
5. Index-Sequential I/O

B.3 IDENTIFICATION DIVISION

1. All CAVS programs and subprograms will have PROGRAM-ID's of six characters or less.

2. Special test programs, not included in the final product, may have names longer than six characters, but these names must be unique within the first five characters.

3. DATE-COMPILED will be used to insert the compile date within the compile listing.

B.4 ENVIRONMENT DIVISION

1. SOURCE-COMPUTER will not be used, unless debugging-mode is also in use.

2. OBJECT-COMPUTER will not be used.

3. The SPECIAL-NAMES paragraph will be used. ANSI COBOL requires the use of "WRITE...AFTER/BEFORE ADVANCING" which in turn is required to define a special name for page feeds.

   Computer-specific special names required to implement input and output will be standardized for each computer.

B.5 DATA DIVISION

1. Level 77 data items will not be used. It will probably be removed from future COBOL standards.

2. Data types.
   - Character data will be stored as ASCII.
   - "USAGE IS INDEX" will be used in limited cases. Source code translated from FORTRAN will not use it.
   - Most numeric information will be stored as full-word integers. The Honeywell data definition is "USAGE IS COMPUTATIONAL". The Univac data definition is "PIC S9(10) USAGE IS COMPUTATIONAL". Sign position will be represented according to the default for each machine.

3. Renames will not be used.

B.6 PROCEDURE DIVISION

1. Logical statements will use the spelled-out logical operators instead of the symbols. For example CAVS will use "EQUALS" instead of "=".

2. Computation and assignment statements.

   - CAVS does not perform any complicated calculations; most computations increment or decrement subscripts as tables are manipulated.

   - Numerical assignment will be done by means of COMPUTE, ADD, and SUBTRACT statements.

   - Simple alphanumeric assignments will be done by using "MOVE".

   - Special calculations will be required to compute the CPU time consumed during testing. These calculations use machine-specific data formats and formulae, and will be isolated in subprograms.

4. Statements not used:
   - ALTER
   - DIVIDE
   - MULTIPLY
   - CORRESPONDING forms of ADD, SUBTRACT, MULTIPLY, DIVIDE, and MOVE, such as MOVE CORRESPONDING.

5. Statements which will be used are
   - ACCEPT
   - ADD
   - CALL

- DISPLAY
- EXIT
- GOTO (in code produced by the precompiler)
- IF
- MOVE
- PERFORM
- SET
- SUBTRACT
- STOP

6. The VAX-11 COBOL supports structured programming constructs which delimit conditional expressions. Examples of such constructs are END-IF and END-PERFORM. The CAVS contract calls for the use of the DMA PSL Structured COBOL precompiler. Because the CAVS contract requires use of the precompiler, and because Univac and Honeywell do not support native COBOL structured programming constructs, the VAX constructs (except for END-IF) will not be used.

   The DMA precompiler constructs to be used are:

   - CASE
   - CASENTRY
   - ELSECASE
   - ENDCASE
   - DO
   - DO UNTIL
   - DO WHILE
   - ENDDO
   - IF
   - ENDIF

7. Statements which may be used in isolated, data-editing contexts are:

- INSPECT
- STRING
- UNSTRING

B.7 TABLE HANDLING

1. Index data items will be used to improve performance in restricted contexts.

2. SET will be used to manipulate index data items and to convert them to printable and display formats.

3. Both sequential and binary searches using the SEARCH verb will be used.

B.8 GENERAL I/O CONSIDERATIONS

1. Blocking and buffering of CAVS data files will be established and tuned for each installation.

2. Honeywell I/O will be controlled by the Unified File Access System (UFAS) that is native to the COBOL-74 compiler.

3. The implementor-name field of the ASSIGN clause in the SELECT statement will be six characters, the first two of which must be unique.

4. Where possible, the actual file name of files used will be assigned and controlled by the job control. Matching of the external physical file with the internal logical file name will be done by means of the ASSIGN clause.

The VALUE-OF-ID clause will be used to specify the physical file name only when it must be dynamically assigned or changed.

5. The file status keys will be used to diagnose the result of all file I/O statements.

6. Indexed Sequential files will be not used.

7. All files except for tape files used for CAVS source installation will use standard labels.

8. Checkpoint/restart will not be used.

B.9 SEQUENTIAL I/O

1. Printer output and pagination will be controlled by counting lines. LINEAGE will not be used.

2. CAVS will read and write COBOL source in ASCII. If the final input or output should be in Fieldata, the Univac System utility @ FURPUR will be used to convert them.

3. Programs instrumented by CAVS will produce an additional sequential output file, the execution trace file. Numeric data in this file will be stored as full word binary integers. Character data will be stored as ASCII.

B.10 RELATIVE I/O

1. DELETE will not be used.

2. CAVS will use ASCII representation for all internal data manipulation and I/O.

3. Variable-length records will not be used.

B.11 LIBRARY

1. The COBOL COPY feature will be used.

2. The COPY REPLACING feature may be used to simplify the conversion of CAVS from one vendor to another.

B.12 DEBUG

1. Debug statements will be used.

B.13 INTERPROGRAM COMMUNICATION

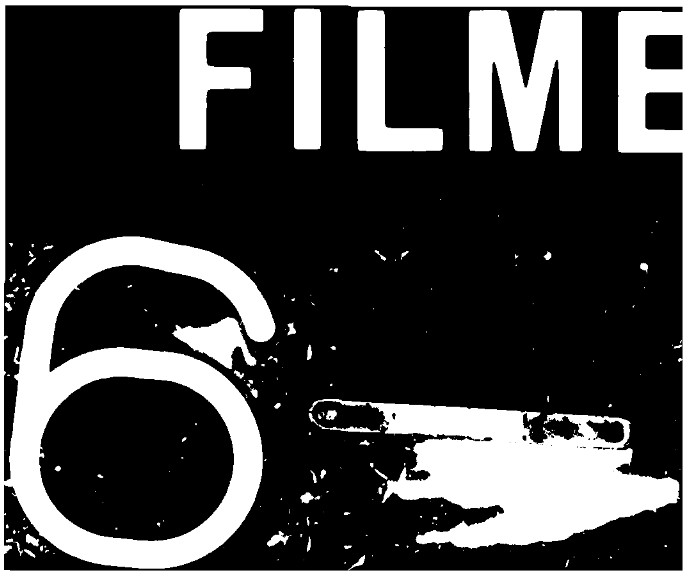
1. APPLY (in Univac and VAX) will not be used.

2. The Common-Storage Section, while not standard COBOL, is available on the Univac compiler, and will be used to conserve memory.

3. Multiple entry points to a subprogram will not be used.

4. The CANCEL statement will be used.

5. ENTER will not be used.


FILME
6